Programming with Purity Reflection: Peaceful Coexistence of Effects, Laziness, and Parallelism

Magnus Madsen
Department of Computer Science, Aarhus University, Denmark

Jaco van de Pol
Department of Computer Science, Aarhus University, Denmark

Abstract

We present purity reflection, a programming language feature that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. The upshot is that operations on data structures can selectively use lazy and/or parallel evaluation while ensuring that side effects are never lost or re-ordered. The technique builds on a recent Hindley-Milner style type and effect system based on Boolean unification which supports both effect polymorphism and complete type inference. We illustrate that avoiding the so-called 'poisoning problem' is crucial to support purity reflection.

We propose several new data structures that use purity reflection to switch between eager and lazy, sequential and parallel evaluation. We propose a DelayList, which is maximally lazy but switches to eager evaluation for impure operations. We also propose a DelayMap which is maximally lazy in its values, but also exploits eager and parallel evaluation.

We implement purity reflection as an extension of the Flix programming language. We present a new effect-aware form of monomorphization that eliminates purity reflection at compile-time. And finally, we evaluate the cost of this new monomorphization on compilation time and on code size, and determine that it is minimal.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases type and effect systems, purity reflection, lazy evaluation, parallel evaluation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.18

1 Introduction

Programming languages are increasingly multi-paradigm. Kotlin and Scala embrace object-oriented, functional, and imperative programming. JavaScript has a functional core and its ecosystem is increasingly adopting a functional style. Rust, a decidedly imperative language, has a functional flavor with support for algebraic data types, pattern matching, and higher-order functions. C# and Java have adopted lambda expressions and added streams.

Nevertheless, the marriage of paradigms is not always a happy one: laziness and parallelism expose a deep rift between functional and imperative programming. The delayed or parallel evaluation of an impure function may cause its side effects to be lost, to occur out-of-order, or to interfere with each other, leading to potentially disastrous consequences. For these reasons, imperative programming languages tend to use eager and sequential semantics everywhere, thus foregoing the potential benefits of lazy and/or parallel evaluation.

Most mainstream languages, such as Java, Kotlin, and Scala, offer access to a limited form of laziness and parallelism with streams. Yet anarchy reigns: the use of side effects in streams can have unpredictable consequences and nothing prohibits stream operations from having side effects, except for stern warnings in the documentation. Stream pipelines are often described as declarative, but in the presence of side effects, they are anything but that.

We propose to overcome these challenges with a new programming language construct that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. For example, List.map can vary its behavior

as follows: when given a *pure* function it lazily maps the function over the list, whereas when given an *impure* function it eagerly maps the function over the list. Thus, List.map ensures that side effects are never lost or re-ordered while simultaneously allowing lazy evaluation for pure functions. We say that the List.map function is *purity reflective*. Similarly, Set.count can vary its behavior as follows: when given a *pure* function it performs the counting in parallel over the set, whereas when given an *impure* function it performs the counting sequentially following the order of the elements in the set. Thus, Set.count ensures that side effects do not lead to thread-safety hazards (like deadlocks, race conditions), while still admitting parallel evaluation when given a pure function. Purity reflection empowers programmers, and in particular library authors, to write new data structures that selectively use lazy and/or parallel evaluation under the hood, while semantically appearing to their clients *as-if* always under eager, sequential evaluation.

We argue that purity reflection, as a simple form of effect reflection [22], hits a "sweet spot" for practical programming. The distinction between pure and impure functions is straightforward and understandable by ordinary programmers, while at the same time providing sufficient information to be useful. We want to stress that purity reflection *increases the value of effect systems*. In particular, most use cases of type and effect systems focus on soundness, i.e. the ability to rule out certain erroneous programs. This is of course very desirable, but it does not really add any new expressive power to a programming language. With purity reflection (and effect reflection in general), we empower programmers to write new programs that they could not express before. Thus, "fighting the types and effects" now comes with an additional reward. Purity reflection is enabled by a recent technique to infer *fine-grained*, polymorphic effects automatically [27].

In this paper, we implement purity reflection, from end-to-end, in a production compiler. We use the implementation to retrofit existing and implement new data structures. A key implementation technique is the use of an effect-aware form of monomorphization. In theory, this technique could lead to an exponential blow-up in compilation time and code size, but we experimentally show that this is not the case.

In summary, the contributions of this paper are:

- (**Purity Reflection**) We introduce purity reflection, a new programming language feature that enables higher-order functions to inspect the purity of their function arguments and to vary their behavior based on this information. We argue that purity reflection is a "sweet spot" in the design space of effect reflection.
- (**Data Structures**) We propose several new data structures that use purity reflection to switch between lazy and eager, sequential and parallel evaluation, including the DelayList and DelayMap data structures.
- (**Compilation**) We discuss two compilation strategies supporting purity reflection: one based on extending the runtime to track purity information in closures and the other based on a new form of effect-aware monomorphization. We implement the latter.
- (**Implementation**) We extend the Flix programming language with purity reflection. We believe Flix is the first large-scale programming language development to support any form of effect reflection.
- (**Evaluation**) We experimentally evaluate the impact of effect-aware monomorphization on compilation time and code size. The results show that the overhead is minimal.

2 Motivation

We motivate our idea with an example. We will use the Flix programming language, but our technique is equally applicable to other ML-style programming languages.

2.1 A Word & Line Count Program

Imagine that we want to write a program that determines if a text contains a specific word. We might start with the program fragment:

```
use List.{flatMap, memberOf};
use String.splitOn;
let lines = haystack |> splitOn("\n");
let words = lines |> flatMap(l -> splitOn(" ", l));
memberOf(needle, words)
```

The program works as follows: Given two strings: haystack and needle, the program splits haystack into a list of lines, then it flatMaps over each line splitting it into a list of words, and finally it computes if words contains the string needle.

The program works as expected and is written in a natural style: We have two local variables: lines and words that hold understandable intermediate results. Unfortunately, the program is not very efficient. We construct several intermediate lists and these entire lists are not even needed if the search word needle occurs early in the text.

If evaluation of splitOn and flatMap were lazy, then the program would run fast and not require the construction of these large intermediate lists. Instead, splitOn and flatMap would build and operate on lazy lists, whose elements would be constructed on-demand when needed by memberOf. *But*, since Flix is strict, this is not the case at the moment.

Let us imagine that we later decide to extend the program to also count the number of lines and words in the text, reminiscent of the wc command from UNIX. Thus, we change the program to:

```
let lineCount = ref 0;
let wordCount = ref 0;
let lines = haystack |> splitOn("\n");
let words = lines |> flatMap(l -> {
    lineCount := deref lineCount + 1;
    let ws = splitOn(" ", l);
    wordCount := deref wordCount + length(ws);
    ws
});
println("Lines: ${deref lineCount}");
println("Words: ${deref wordCount}");
println("Found: ${memberOf(needle, words)}")
```

The extended program is more sophisticated. Running it might print: `Lines:  21, Words:  261, Found:  true`. The new program uses a natural style of functional and imperative programming that is common in Java, Kotlin, and Scala. The core of the program remains functional, but the counting is performed in an imperative manner. The program could be written in a purely functional style, but this would require careful threading of state: We would have to operate on triples of the current words on a line and the two counters.

Importantly, this program must be evaluated eagerly. If we were to lazily evaluate splitOn and flatMap then the two counters would not be updated before they are printed, and the program would print the wrong result (e.g. `Lines:  0, Words:  0, true`).

The two programs expose a rift between functional and imperative programming. In the functional paradigm we would like certain operations to be lazy to improve performance, whereas in the imperative paradigm it is vital that effectful operations are evaluated eagerly.

We believe that the fundamental tension is between two different views on the essence of operations such as filter, map, and flatMap. In the *imperative view*, these operations *eagerly* transform one data structure into another data structure. If the transformation has any side effects, these occur immediately and in a deterministic order (e.g. the order of a list, the natural order of a tree, etc.). In the *declarative view*, these operations *describe* how a data structure should be transformed, but the transformation is not applied until *needed*. In this view, effectful transformations are evil; either banned outright (like in Haskell) or strongly discouraged with stern warnings (like in Java, Scala).

So what can be done? In this paper, we propose a technique, i.e. purity reflection, where we can have our cake and eat it too. Purity reflection allows both programs – *exactly as written* – to compute their expected results while the first is evaluated lazily and the second is evaluated eagerly. This allows us to *write programs the way we want* while ensuring that side effects are never lost and always occur in the expected order.

We can use purity reflection to switch between eager and lazy evaluation, but our technique is equally applicable to switching between sequential and parallel evaluation. For example, if we know that the predicate function passed to Set.count is pure, then it is safe to evaluate the function in parallel over disjoint subsets of the set.

2.2 Streams: An Unsound Solution

Before we proceed, we want to highlight the challenges posed by trying to combine side effects, laziness, and parallelism in a single programming language. Mainstream programming languages, such as Java and Scala, support a small collection of data structures that are lazy and/or parallel. Most prevalent is the support for *streams*, a lazy (and sometimes parallel) data structure that represents a sequence of elements.

2.2.1 Java

The java.util.Stream package offers a collection of utilities for working with "sequences of elements supporting sequential and parallel aggregate operations". The documentation for the package states that[1]:

> *"side effects in behavioral parameters to stream operations are, in general, discouraged, as they can often lead to unwitting violations of the statelessness requirement, as well as other thread-safety hazards."*

A bit later, the documentation goes on to state:

> *"[...] The ordering of side effects may be surprising. [...] The eliding of side effects may also be surprising. [...]"*

In total, the documentation for Stream warns about side effects almost twenty times!

[1] https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/stream/package-summary.html

2.2.2 Scala

The scala.collection.parallel package offers a collection of parallel data structures. The documentation for the package states that[2]:

> *"[...] These concurrent and "out-of-order" semantics of parallel collections lead to the [...] implications:*
> - *Side effecting operations can lead to non-determinism*
> - *Non-associative operations lead to non-determinism*
>
> *Given the concurrent execution semantics of the parallel collections framework, operations performed on a collection which cause side effects should generally be avoided, in order to maintain determinism."*

The documentation for ParIterable goes on to state[3]:

> *"[...] Since implementations of bulk operations may not be sequential, this means that side effects may not be predictable and may produce data-races, deadlocks or invalidation of state if care is not taken. [...]*

As these examples illustrate, the combination of effectful operations with lazy and/or parallel evaluation is fraught with danger. A mindful programmer is left weary. Perhaps as a consequence, a study by Khatchadourian et al. finds that "stream parallelization is rarely used", despite the fact that "streams tend not to have side effects" [17].

This paper provides a path out of the quagmire.

2.3 Proposed Solution

We propose a solution based on the following simple idea:

> Data structure operations (such as map, filter, ...) may use lazy or parallel evaluation when they are given pure function arguments, but revert to eager, sequential evaluation when given impure function arguments to ensure that side effects are not lost and that the order of effects is preserved.

We illustrate this idea with two examples:

```
@LazyWhenPure                          @ParallelWhenPure
def map(f: a -> b & ef, l:List[a])     def cnt(f: a -> Bool & ef, s:Set[a])
    = reifyEff(f) {                        = reifyEff(f) {
        case Pure(g) => mapL(g, l)             case Pure(g) => parCnt(g, s)
        case _       => mapE(f, l)             case _       => fold(...)
    }                                      }
```

The program construct `reifyEff(exp)` allows us to reflect on the purity of the closure exp. In the program fragment on the left, which implements the map function on a list, we use reifyEff to inspect the purity of the function argument f. If it is pure, we use mapL to lazily apply the function f over the list (i.e. no evaluation happens yet). If, on the other hand, f is impure we use mapE to immediately apply f eagerly over the elements of the list.

[2] https://docs.scala-lang.org/overviews/parallel-collections/overview.html
[3] https://www.scala-lang.org/api/2.12.2/scala/collection/parallel/ParIterable.html

Note that in the pure case, reifyEff rebinds f as g (i.e. it is the same function), but now g is typed as a pure function. This rebinding avoids the need for flow-sensitive typing.

In the program fragment on the right, we use reifyEff to determine whether to count the elements that satisfy a given predicate f sequentially or in parallel over a set. If f is pure then we perform the counting in parallel, otherwise, we perform it sequentially using an ordinary fold.

The annotations @LazyWhenPure and @ParallelWhenPure have no semantic meaning, but serve as documentation for the programmer.

The reifyEff construct is enabled by a recent Hindley-Milner style type and effect system that supports effect polymorphism, type inference, and computes purity information for every sub-expression in a program [27]. Building on this type and effect system, we can implement purity reflection as a compile-time programming construct that is eliminated by a new form of effect-aware monomorphization. For example, we will monomorph two versions of the map function: one for pure functions and one for impure functions.

We now discuss some properties of the proposed solution:

- (*Modular*) The technique supports abstraction: A library *author* can implement data structure operations that make selective use of lazy or parallel evaluation without leaking those details to the client. A library *user* can reason about his or her code *as-if* under eager and sequential semantics.
- (*Gradual*) It is easy to start using the technique: A data structure can be made gradually lazy or parallel without affecting the semantics of its clients†.
- (*Programmable*) The technique is based on a new programming language construct. Thus, maximum power is placed in the hands of library authors (and programmers in general) who may have better knowledge of when to exploit laziness or parallelism.
- (*Zero Cost*‡) The new programming construct can be eliminated entirely at compile-time. Thus programs using the technique suffer no runtime overhead.
- (*Sound*††) The technique is based on a sound type and effect system: It ensures that if an expression is pure then it cannot have a side effect. The typing of lazy expressions ensures that side effects cannot be hidden and later revealed.

† Of course, programmers and library authors should be aware that (i) switching from eager to lazy evaluation can potentially lead to space leaks, and (ii) switching from sequential to parallel evaluation may slow down the program. However, we believe both situations can be managed. For (i), lazy evaluation should only be used for stream-like data structures where space leaks are less likely to occur, and for (ii), parallel evaluation should use light-weight threads and only be enabled for sufficiently large data structures.

‡ We use an effect-aware form of monomorphization that specializes (i.e. copies) higher-order functions based on the purity of their function argument(s). This ensures that there is no runtime overhead, but it could potentially lead to increased compilation time and increased code size. In Section 7 we experimentally evaluate this cost.

†† The technique does not magically guarantee correctness. For example, a programmer could mistakenly implement List.map to always return the empty list when given a pure function argument. This does not violate the soundness of the type and effect system itself, but it does violate the commonly understood specification of what List.map should do.

$$
\begin{array}{rcl}
c \in \mathit{Cst} & = & () \mid \mathsf{true} \mid \mathsf{false} \mid \cdots \\
v \in \mathit{Val} & = & c \mid \lambda x.\, e \\
e \in \mathit{Exp} & = & x \mid v \mid e\, e \\
& \mid & \mathsf{let}\, x = e\, \mathsf{in}\, e \\
& \mid & \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
& \mid & \mathsf{lazy}\ e \mid \mathsf{force}\ e \\
& \mid & \mathsf{print}\, e \\
x, y \in \mathit{Var} & = & \text{a set of variables}
\end{array}
$$

(a) Expressions of $\lambda_{\mathcal{B}}$.

$$
\begin{array}{rcl}
\tau \in \mathit{Type} & = & \alpha \mid \iota \mid \tau \xrightarrow{\varphi} \tau \mid \mathsf{lazy}\ \tau \\
\varphi \in \mathit{Formula} & = & \mathtt{T} \mid \mathtt{F} \mid \beta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \\
\sigma \in \mathit{Scheme} & = & \tau \mid \forall\alpha.\, \sigma \mid \forall\beta.\, \sigma \\
\\
\iota \in \mathit{BaseType} & = & \mathsf{Unit} \mid \mathsf{Bool} \mid \mathsf{Int} \mid \cdots \\
\alpha \in \mathit{TypeVar} & = & \text{a set of type variables} \\
\beta \in \mathit{BoolVar} & = & \text{a set of Boolean variables}
\end{array}
$$

(b) Types of $\lambda_{\mathcal{B}}$.

Figure 1 Syntax and Types of $\lambda_{\mathcal{B}}$.

3 Purity Reflection

We begin with a brief introduction to the $\lambda_{\mathcal{B}}$ calculus and its Hindley-Milner-style type and effect system [12, 30, 7]. The system is from [27] but extended with the standard lazy and force constructs [31]. The $\lambda_{\mathcal{B}}$ calculus is the foundation for the Flix programming language implementation (which we build on top of). The $\lambda_{\mathcal{B}}$ calculus is proven sound in [27]. In Section 3.4, we propose a simple extension that requires just one new expression and one new type rule.

3.1 A Minimal Calculus

Syntax

The syntax of $\lambda_{\mathcal{B}}$ includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function application. As is standard in Hindley-Milner style type systems, the let-expression $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ supports polymorphic generalization of e_1[4]. The if-then-else expression is standard and included to illustrate how the type and effect system merges information from different control-flow paths. The print expression is included to have a side effect in the calculus. We add lazy e and force e to suspend and resume computations. We assume that force e uses memoization. Figure 1a shows the syntax of $\lambda_{\mathcal{B}}$.

Semantics

We assume a standard call-by-value semantics, i.e., function arguments are reduced to values before they are substituted into the body of a lambda abstraction. The same applies to let-bindings. The only exceptions are `if-then-else`, which uses short circuiting semantics, and `lazy` expressions, which are treated as closures that are computed only once when forced, and then memoized.

[4] In Flix, which has mutable references, let-generalization is subject to the value restriction [8].

$$\frac{\Gamma \vdash e : \tau_1 \,\&\, \varphi_1 \quad \tau_1 \equiv \tau_2 \quad \varphi_1 \equiv \varphi_2}{\Gamma \vdash e : \tau_2 \,\&\, \varphi_2}\ (\text{T-Eq}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varphi_1 \quad \Gamma \vdash e_2 : \tau_1 \,\&\, \varphi_2}{\Gamma \vdash e_1\, e_2 : \tau_2 \,\&\, \varphi_1 \wedge \varphi_2 \wedge \varphi}\ (\text{T-App})$$

$$\frac{\mathsf{typeOf}(c) = \sigma \quad \sigma \sqsubseteq \iota}{\Gamma \vdash c : \iota \,\&\, \mathtt{T}}\ (\text{T-Cst}) \qquad \frac{\Gamma \vdash e : \tau \,\&\, \mathtt{T}}{\Gamma \vdash \mathsf{lazy}\ e : \mathsf{lazy}\ \tau \,\&\, \mathtt{T}}\ (\text{T-Lazy})$$

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau \,\&\, \mathtt{T}}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash e : \mathsf{lazy}\ \tau \,\&\, \varphi}{\Gamma \vdash \mathsf{force}\ e : \tau \,\&\, \varphi}\ (\text{T-Force})$$

$$\frac{\Gamma \vdash e : \mathsf{String} \,\&\, \varphi}{\Gamma \vdash \mathsf{print}\, e : \mathsf{Unit} \,\&\, \mathtt{F}}\ (\text{T-Prt}) \qquad \frac{\Gamma \vdash e_1 : \tau_1 \,\&\, \varphi_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2 \,\&\, \varphi_1 \wedge \varphi_2}\ (\text{T-Let})$$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2 \,\&\, \varphi}{\Gamma \vdash \lambda x.\, e : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \mathtt{T}}\ (\text{T-Abs}) \qquad \frac{\Gamma \vdash e_1 : \mathsf{Bool} \,\&\, \varphi_1 \quad \Gamma \vdash e_2 : \tau \,\&\, \varphi_2 \quad \Gamma \vdash e_3 : \tau \,\&\, \varphi_3}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau \,\&\, \varphi_1 \wedge \varphi_2 \wedge \varphi_3}\ (\text{T-Ite})$$

$$\mathsf{gen}(\Gamma, \tau) = \forall \alpha_1, \cdots, \forall \alpha_n . \forall \beta_1, \cdots, \forall \beta_n . \tau \quad \text{where} \quad \{\alpha_1, \cdots, \alpha_n, \beta_1, \cdots, \beta_n\} = \mathsf{ftv}(\tau) \setminus \mathsf{ftv}(\Gamma)$$

Figure 2 Type Rules for $\lambda_{\mathcal{B}}$ with judgments of the form $\Gamma \vdash e : \tau \,\&\, \varphi$.

3.2 Type and Effect System

Types

The types of $\lambda_{\mathcal{B}}$ are separated into monotypes (τ) and type schemes (σ). The monotypes include type variables α, a set of base types ι, and function types $\tau_1 \xrightarrow{\varphi} \tau_2$ that represents functions from values of type τ_1 to values of type τ_2 with *latent* effect φ. We use the type $\mathsf{lazy}\ \tau$ to denote suspended computations. The type schemes of $\lambda_{\mathcal{B}}$ include monotypes τ and quantified types $\forall\alpha.\sigma$ and $\forall\beta.\sigma$, where α is a type variable and β is a Boolean effect variable. Figure 1b shows the types and type schemes of $\lambda_{\mathcal{B}}$.

In $\lambda_{\mathcal{B}}$ the language of effects is a single Boolean formula φ, i.e. there is only a single "effect": impurity. If the Boolean formula is equivalent to true ($\mathtt{T}$) then the expression it describes must be pure. If the Boolean formula is equivalent to false ($\mathtt{F}$) then the expression may have a side effect. A Boolean formula with variables in it captures the conditions under which the expression is pure. The system is over-approximating: An expression typed as pure *cannot* have a side effect whereas an expression typed as impure *may* have a side effect [27].

Type Judgements

Figure 2 shows the type rules of $\lambda_{\mathcal{B}}$. We define a context Γ as a *partial function* of bindings $x : \sigma$ from variables to type schemes. We also define $\mathsf{ftv}(\sigma)$ to be the type variables that occur free in σ, and $\mathsf{ftv}(\Gamma)$ as the union of all free type variables in its range. A type judgement is of the form $\Gamma \vdash e : \tau \,\&\, \varphi$, which states that under type environment Γ, the expression e has type τ and effect φ, where φ is a Boolean formula that captures when the expression is pure.

We now briefly discuss the most important type rules. Except for (T-Eq), the rules are syntax-directed. The (T-Cst) rule states that a constant expression is pure. The (T-Ite) rule states in an $\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3$ the overall effect is $\varphi_1 \wedge \varphi_2 \wedge \varphi_3$ where φ_i is the effect of expression e_i. The (T-Abs) and (T-App) rules type lambda abstractions and applications. An abstraction takes the effect φ of an expression e and moves it onto the arrow type whereas an application releases the latent effect of the arrow type. The (T-Var) and (T-Let) rules are the standard Hindley-Milner type rules. We add the (T-Lazy) and (T-Force) rules. Note that only pure expressions can be suspended. Thus effects cannot be delayed. The (T-Eq) rule states that we can use type equivalence. In $\lambda_{\mathcal{B}}$ two types are considered equivalent modulo Boolean equivalence.

For example, the following two functions types are equivalent:

$$\mathsf{Int} \xrightarrow{x \vee \neg x} \mathsf{Int} \quad \equiv_{\mathbb{B}} \quad \mathsf{Int} \xrightarrow{\mathtt{T}} \mathsf{Int}$$

By a suitable extension of Algorithm W with Boolean unification, the type and effect system supports complete type inference. We refer to [27] for the full details.

3.3 Effect Polymorphism

The $\lambda_{\mathcal{B}}$ calculus supports effect polymorphism, i.e. the effect of a higher-order function may depend on the effects of its function arguments. For example, the List.map function can be given the type:

$$\mathsf{List.map} : \forall \alpha_1, \alpha_2, \beta.\, (\alpha_1 \xrightarrow{\beta} \alpha_2) \xrightarrow{\mathtt{T}} \mathsf{List}[\alpha_1] \xrightarrow{\beta} \mathsf{List}[\alpha_2]$$

which can be read as: the effect of List.map is the same as the effect of its function argument, i.e. List.map is pure if its function argument is.

Forward function composition » can be given the type:

$$» : \forall \alpha_1, \alpha_2, \alpha_3, \beta_1, \beta_2.\, (\alpha_1 \xrightarrow{\beta_1} \alpha_2) \xrightarrow{\mathtt{T}} (\alpha_2 \xrightarrow{\beta_2} \alpha_3) \xrightarrow{\mathtt{T}} (\alpha_1 \xrightarrow{\beta_1 \wedge \beta_2} \alpha_3)$$

which can be read as: the composition of f and g is pure if both are pure. Note that the purity of » is constructed from the purity of both f and g.

3.4 Purity Reflection with ReifyEff

We extend the $\lambda_{\mathcal{B}}$ calculus with a single new expression:

$$\mathsf{reifyEff}(e_1)\{\mathsf{case\ Pure}(f) \Rightarrow e_2, \mathsf{case\ \_} \Rightarrow e_3\}$$

The idea is that if e_1 evaluates to a pure function v then it is bound to f and the whole reifyEff expression reduces to $e_2[f \mapsto v]$. Otherwise, the expression reduces to e_3. Of course, one cannot in general determine whether a function is pure. Thus, we rely on the type and effect system to tell us whether a function value is pure. In other words, in the extended $\lambda_{\mathcal{B}}$ calculus (and Flix in general), only well-typed terms have an operational semantics [35]. In Section 5 we discuss two compilation strategies for how to implement the reifyEff construct.

The type rule for the reifyEff expression is straightforward and shown in Figure 3. The type rule requires that the expression e_1 has a function type $\tau_1 \xrightarrow{\varphi} \tau_2$ where φ is the latent effect of the function. We type check e_2 in an extended environment, where we introduce a new binder f for the function which is typed as pure (i.e., with effect T). We *do not* introduce a new binder for the case where the function is impure. This asymmetry is for two reasons:

- The type and effect system is over-approximating: If an expression is pure then it cannot have a side effect, but the opposite is not true: an impure expression is not guaranteed to produce a side effect.

$$\frac{\Gamma \vdash e_1 : \tau_1 \xrightarrow{\varphi} \tau_2 \,\&\, \varphi_1 \qquad \Gamma, f : \tau_1 \xrightarrow{\mathtt{T}} \tau_2 \vdash e_2 : \tau_3 \,\&\, \varphi_2 \qquad \Gamma \vdash e_3 : \tau_3 \,\&\, \varphi_3}{\Gamma \vdash \mathsf{reifyEff}(e_1)\{\mathsf{case\ Pure}(f) \Rightarrow e_2, \mathsf{case\ \_} \Rightarrow e_3\} : \tau_3 \,\&\, \varphi_1 \wedge \varphi_2 \wedge \varphi_3} \text{ (T-Reify-Eff)}$$

Figure 3 Type rule for the reifyEff construct.

- *If* we had introduced a new binder for f and given it the effect impure (i.e., effect `F`) then any use of f inside e_3 would cause the whole expression to be impure. But this would prevent purity reflection. We would not be able to use reifyEff inside List.map while keeping it effect-polymorphic.

3.4.1 Correctness

The correctness of the approach depends on the soundness of the type and effect system and completeness of type inference [27]. There, it is proved that the type system enjoys the progress- and preservation-property. Also, Algorithm W extended with Boolean unification will always compute the most general type and effect. Finally, it is proved that a pure expression can at no time perform an effectful step. So it is not possible to hide effects. Moreover, the `lazy` construct in Flix can only be applied to pure expressions, so it impossible to delay effects.

Of course, nothing in the type and effect system ensures that an implementation satisfies its specification. For example, a programmer could accidentally reverse a list before mapping an effectful function over it. In that case, the effects will happen in the wrong order. Ensuring *functional correctness*, i.e., that a function respects its specification of return values and emitted effects, is generally beyond the scope of Hindley-Milner style type systems.

3.5 Fine-Grained Purity and the Poisoning Problem

We stress that the type rules are *compositional* and *fine-grained*: the purity of an expression is constructed from the purity of its sub-expressions. This is in contrast to the situation in row-polymorphic type and effect systems [39, 20], where the effects of the sub-expressions are required to be the same. Such systems suffer from the so-called *poisoning problem* [40], where the effect of a sub-expression is over-approximated to fit its context.

We illustrate this issue with an example:

```
airplanes |>
    List.map(plane -> plane.pilot) |>
    List.map(pilot -> pilot.name) |>
    List.foreach(println)
```

Here a row polymorphic system infers that sub-expression `List.foreach(println)` has the Print effect. This in turn pollutes every sub-expression with the Print effect. Consequently, the row polymorphic system cannot be used to infer that the first two List.map operations are pure (and could be applied lazily with our technique). However, the Boolean effect system can infer that each of the List.map operations is given pure arguments, even through the polymorphic usage of the pipeline function `|>`. This example demonstrates that for purity reflection to work, one needs a *compositional* and *fine-grained* type and effect system.

3.6 Purity Reflection: A Sweet Spot

We believe that purity reflection hits a "sweet spot". First, it is simple to explain to programmers: they only have to understand the distinction between pure and impure functions. Second, it requires us to maintain minimal information to implement, either at runtime or compile-time, as discussed in Section 5. Third, as we argue below, a richer effect system may be difficult to exploit in practice. In particular, it is difficult to determine when two effects may interfere. For example:

- (**Aliasing**) Given two effects $\textsc{Read}(p_1)$ and $\textsc{Write}(p_2)$ where p_1 and p_2 are pointers to mutable memory, can we safely evaluate them lazily or in parallel? The answer depends on whether p_1 and p_2 are aliased, i.e., can point to the same memory location. If they are, then any re-ordering or parallel execution may change the meaning of the program. Unfortunately, we cannot statically know if p_1 and p_2 are aliases without additional heavy machinery: either alias analysis or a sub-structural type system. To solve this, one needs more information, such as fine-grained regions [15, 37, 10].
- (**External Aliasing**) Given two effects $\textsc{ReadFile}(f_1)$ and $\textsc{WriteFile}(f_2)$ where f_1 and f_2 are file paths, can we safely evaluate them lazily or in parallel? As before, the answer depends on whether f_1 and f_2 refer to the same file. We cannot statically determine if f_1 and f_2 may denote the same filename without some notion of control- and data flow analysis. Worse, even, if f_1 and f_2 are guaranteed to be distinct strings, the two file paths may still refer to the same file due to symbolic links in the underlying file system.
- (**Implicit Dependencies**) Given two effects WriteFile and CurrentTime, can we safely evaluate them lazily or in parallel? Maybe, but not if the programmer is trying to measure the time it takes for the WriteFile operation to complete.

These examples do not imply that the task is impossible. If we had a specification of each effect, i.e., if we had much more information from the programmer, we could probably apply lazy and/or parallel evaluation more aggressively. Instead, our system makes the simple and sound assumption that effects should never be omitted nor re-ordered.

4 Four New Data Structures

We now illustrate how reifyEff can be used to extend two existing and implement two new data structures that make selective use of lazy and/or parallel evaluation.

4.1 From List to LazyList to DelayList

4.1.1 From List to LazyList

In Flix, the familiar definition of List is:

```
enum List[a] {
    case Nil,
    case Cons(a, List[a])
}
```

A list is either the empty list Nil or a cons cell Cons(x, xs) with an element x and a tail xs. We can implement list operations such as filter, map, and flatMap in the standard way.

The definition of List does not permit lazy evaluation. We can fix that by redefining List to have a lazy tail:

```
enum List[a] {
    case Nil,
    case Cons(a, Lazy[List[a]])
}
```

Flix has two expressions to support lazy evaluation: lazy e and force e. The former suspends the evaluation of an expression e returning a thunk of type Lazy[t] where t is the type of the expression. The latter evaluates a thunk and memoizes the result. Recall that only pure expressions can be suspended. With the updated definition of List, we can express eager and lazy versions of every list operation.

For example, here is the lazy definition of map[5]:

```
def mapL(f: a -> b, l: List[a]): List[a] =
    match l {
        case Nil          => Nil
        case Cons(x, xs) =>
            // The tail is *not* yet evaluated.
            Cons(f(x), lazy mapL(f, force xs))
    }
```

The mapL function takes a pure function f from values of type a to type b and a list l of elements of type a. If the list is empty, it returns the empty list. Otherwise, there is a Cons(x, xs) cell where the tail xs is lazy. In this case, we evaluate f on the head x and construct a lazy computation that maps f over the rest of the list xs. Thus, no evaluation of the tail happens until it is needed. Note that the use of `lazy` and `force` requires the suspended computation to be pure. Consequently, f must be pure, as reflected in the function signature.

We can also implement an eager and *effect-polymorphic* version of map[6]:

```
def mapE(f: a -> b & ef, l: List[a]): List[a] & ef =
    match l {
       case Nil          => Nil
       case Cons(x, xs) =>
          let hd = f(x);                 // Eagerly evaluate f(x)
          let tl = mapE(f, force xs);    // Force the rest of the list
          Cons(hd, lazy tl)       // Tail is lazy, but fully evaluated
    }
```

The mapE function takes a function f from values of type a to type b with latent effect ef and a list l of elements of type a. It pattern-matches on l. If the list is empty it returns the empty list. Otherwise, there is a Cons(x, xs) cell where the tail xs is lazy. We evaluate f on the head x; then we perform a recursive call on the tail xs (forcing the list). Note that moving the tail-computation to a let-binding makes it eager. Finally, we return a cons-cell with the new head and tail, where the tail is made lazy (but nevertheless has been fully evaluated). Unlike, mapL, the mapE function permits side effects, because it materializes those effects immediately.

We now have mapL and mapE which have lazy and eager semantics, respectively. We can use these two functions to define a *purity reflective* map function that varies its behavior depending on the purity of its function argument:

```
def map(f: a -> b & ef, l: List[a]): List[b] & ef =
    reifyEff(f) {
        case Pure(g) => mapL(g, l)   // Use lazy evaluation.
        case _       => mapE(f, l)   // Use eager evaluation.
    }
```

The implementation is straightforward: The map function matches on the purity of f. If f is pure, then we bind it to g (which is typed as pure) and call mapL passing g. Otherwise, f may be impure, and we call mapE.

---

[5] The syntax `a -> b` denotes a pure function from a to b.

[6] The syntax `a -> b & ef` denotes an effect polymorphic function from values of type a to type b with latent effect ef.

Example I

The Flix program fragment[7]:

```
List.range(1, 1_000_000_000) |>              // Lazy
List.map(x -> {println("a"); x + 1}) |>      // Eager
List.map(x -> {println("b"); x * 2})         // Eager
```

Prints one billion a's followed by one billion b's. This takes a while, but ultimately the program terminates. The a's are printed before the b's preserving the order of effects.

Example II

The following Flix program fragment Prints Some(4) and terminates immediately. The two map operations are pure, consequently they are applied lazily and only evaluated for the first element of the list.

```
List.range(1, 1_000_000_000) |>              // Lazy
List.map(x -> x + 1) |>                      // Lazy
List.map(x -> x * 2) |>                      // Lazy
List.head |> println                         // Eager in head.
```

Example III

The Flix program fragment:

```
let count = ref 0;
List.range(1, 1_000_000_000) |>              // Lazy
List.map(x -> x + 1) |>                      // Lazy
List.take(1_000) |>                          // Lazy
List.map(x -> {                              // Eager
    count := deref count + 1; x * 2
});
println(deref count)
```

Prints 1000 and terminates rather quickly. The first map operation is applied lazily, and the subsequent take operation is also applied lazily. The final map operation is applied eagerly, but only to the first 1000 elements.

4.1.2 From LazyList to DelayList

While the previous lazy list data structure permits both eager and lazy evaluation, its representation is inefficient. In particular, the lazy list definition has two issues: (i) each use of Lazy introduces a layer of indirection. This indirection requires extra memory, slows down access, and puts additional pressure on the garbage collector, and (ii) each force operation is guarded by a lock to ensure that the thunk is evaluated at most once. This can cause lock contention and is antithetical to the idea that immutable data structures can be shared freely and efficiently in a concurrent program. To overcome these issues, we actually use the following definition:

[7] The List.range(b, e) function returns a (suspended) list of integers from b until e.

```
enum DelayList[a] {
    case ENil
    case ECons(a, DelayList[a])
    case LCons(a, Lazy[DelayList[a]])
}
```

The idea is that a fully evaluated list is represented with the ENil and ECons constructors whereas a list with a lazy tail is represented with the LCons constructor. We implement all operations to be maximally lazy. Evaluation occurs for two reasons: (i) when required by a terminal operation (e.g., count and foldLeft), or (ii) when a non-terminal operation is given an impure function argument (e.g., filter and map).

We have implemented the DelayList data structure. The following pure operations are always lazy: append, drop, flatten, from, intercalate, intersperse, range, repeat, replace, take, and zip. The operations shown in Fig. 4a are lazy when given pure function arguments and otherwise eager.

We can use the DelayList data structure to realize the word count example from Section 2.

4.2 From Set to Parallel Set

We have refactored the Flix Standard Library implementation of the Set data structure to use parallel evaluation for all of its aggregation operations (shown in Fig. 4b). Each of these functions use reifyEff to inspect the purity of their function argument, and then they dispatch to either a sequential or a parallel function that operates on the underlying Red-Black tree. For example, here is the implementation of Set.count:

```
@ParallelWhenPure
pub def count(f: a -> Bool & ef, s: Set[a]): Int32 & ef =
    reifyEff(f) {
        case Pure(g) if useParallelEvaluation(s) =>
            RedBlackTree.parCount(g, s)
        case _ =>
            foldLeft((b, x) -> if (f(x)) b + 1 else b, 0, s)
    }
```

Here we use purity reflection on f to determine whether it is safe to perform the count in parallel or if we must perform it sequentially (going from left to right). Moreover, we use the function useParallelEvaluation to estimate whether it is *worth* to perform the count in

@LazyWhenPure

```
def dropWhile(...)
def filter(...)
def filterMap(...)
def flatMap(...)
def map(...)
def mapWithIndex(...)
def span(...)
def takeWhile(...)
def zipWith(...)
```

(a) DelayList: lazy when pure.

@ParallelWhenPure

```
def count(...)
def maximumBy(...)
def minimumBy(...)
def productWith(...)
def sumWith(...)
```

(b) Set: parallel when pure.

@ParallelWhenPure

```
def count(...)
def map(...)
def mapWithKey(...)
def maximumKeyBy(...)
def maximumValueBy(...)
def minimumKeyBy(...)
def minimumValueBy(...)
def productWith(...)
def sumWith(...)
```

(c) Map: parallel when pure.

Figure 4 Selective Lazy *or* Parallel datastructures, depending on purity of function arguments.

parallel. In particular, the useParallelEvaluation function relies on some heuristics, including the height of the Red-Black tree, to determine whether we *should* use parallel evaluation, given that we *could*.

The implementation of RedBlackTree.parCount is straightforward:

```
@Parallel
def parCount(f: (k, v) -> Bool, t: RedBlackTree[k, v]): Int32 =
    match t {
        case Leaf                   => 0
        case DoubleBlackLeaf        => 0
        case Node(_, l, k, v, r)    => // left, key, value, right
            par (cl <- parCount(f, l);
                 cm <- if (f(k, v)) 1 else 0;
                 cr <- parCount(f, r))
            yield cl + cm + cr
    }
```

Here we use the built-in Flix construct `par` to evaluate three expressions in parallel. Thus, the count is performed in parallel on the left subtree, on the key and value, and on the right subtree.

We might worry that spawning too many threads may impose an overhead much larger than the time saved by using parallel evaluation. With OS-level threads, which are expensive, this is likely to be the case. A standard solution to this problem is the use of thread pools and/or a fork-join framework. However, with the imminent arrival of light-weight threads in Java (Project Loom), we hope that such administration will no longer be necessary since VirtualThreads are very cheap.

4.3 From Map to Parallel Map

We have also refactored the Flix Standard Library implementation of the Map data structure (mapping keys to values), to use parallel evaluation for all the aggregation and transformation operations when given pure function arguments (shown in Fig. 4c). The map and mapWithKey functions are the most interesting since they allow parallel rebuilding of the map when applying a pure function to all of its values. As before, these functions use reifyEff to dispatch to the appropriate operation inside RedBlackTree.

For example, here is a simplified version RedBlackTree.parMapWithKey:

```
@Parallel
pub def parMapWithKey(f: (k, v1) -> v2, t: RedBlackTree[k, v1]):
        RedBlackTree[k, v2] =
    match t {
        case Leaf                   => Leaf
        case DoubleBlackLeaf        => DoubleBlackLeaf
        case Node(c, l, k, v, r)    =>
            par (l1 <- parMapWithKey(f, l);
                 v1 <- f(k, v);
                 r1 <- parMapWithKey(f, r))
            yield Node(c, l1, k, v1, r1)
    }
```

4.4 From Map and ParallelMap to DelayMap

We now turn to perhaps the most interesting new data structure: DelayMap, a data structure that uses *both* selective lazy and parallel evaluation. A DelayMap[k, v] is a map from strict keys (of type k) to lazy values (of type v). Pure transformations on the values of a DelayMap

are lazy (e.g., DelayMap.map). Pure terminal operations use parallelism (e.g., Delay.count). The operations in Fig. 5a use lazy evaluation when given pure function arguments. The unionWith operation is especially interesting, as it enables the lazy merge of two maps (if the `combine` operation is pure). The operations in Fig. 5b use parallel evaluation when given pure function arguments. Consider the program fragment:

```
m1 |> DelayMap.map(x -> x + 1)                  // Lazy.
      DelayMap.map(x -> x * 2)                  // Lazy.
      DelayMap.count(x -> {println(x); x > 5}) // Parallel + sequential
```

Assume we start with a map `m1` from strings to integers. The first and second map operation transform the values and are applied lazily because they are pure. The last count operation is impure, hence must be applied eagerly. However, before doing so, we force the entire map in parallel (i.e., applying `x -> x + 1` and `x -> x * 2` to every value) and only then the counting is performed, sequentially from left to right. This example shows that we apply pure operations lazily, but once we have performed an impure operation, we force the entire data structure in parallel, and then switch back to sequential evaluation.

4.4.1 Example I

We can use a DelayMap to write code that is both natural and efficient. Assume that we have two maps m_1 and m_2 of type DelayMap[String, Int32]. Each map records the number of occurrences of a specific word drawn from some documents d_1 and d_2. We can merge the two maps and then compute the total number of occurrences of the word "`foo`":

```
let m1 = ...
let m2 = ...
let m3 = DelayMap.unionWith((x, y) -> x + y, m1, m2);
DelayMap.getWithDefault("foo", 0, m3) |> println
```

Here the unionWith function merges two maps using the supplied merge function to resolve conflicts when a key occurs in both maps. The merge operation is pure and hence unionWith is evaluated lazily. This means that we only have to merge and perform the addition for the key "`foo`" (and any other key we may look up).

@LazyWhenPure

```
def adjust(...)
def adjustWithKey(...)
def insertWith(...)
def insertWithKey(...)
def map(...)
def mapWithKey(...)
def unionWith(...)
def unionWithKey(...)
def update(...)
def updateWithKey(...)
```

(a) DelayMap: lazy when pure.

@ParallelWhenPure

```
def count(...)
def maximumKeyBy(...)
def maximumKeyBy(...)
def maximumValueBy(...)
def minimumKeyBy(...)
def minimumValueBy(...)
def sumWith(...)
def productWith(...)
```

(b) DelayMap: parallel when pure.

Figure 5 Selective Lazy *and* Parallel datastructure, depending on purity of function arguments.

Table 1 Overview of Data Structures. (*LWP* = Lazy When Pure, *PWP* = Parallel When Pure).

Data Structure	Lines	Tests	Functions	`@LazyWhenPure`	`@ParallelWhenPure`	LWP + PWP
Set	610	384	51	-	5	-
Map	924	591	80	-	9	-
DelayList	1,158	498	54	9	-	-
DelayMap	786	298	58	10	9	2

4.4.2 Example II

We can merge the two DelayMaps, while using a mutable list to compute all the words that occur in both maps:

```
let m1 = ...
let m2 = ...
let duplicates = MutList.empty();
let merge = (key, x, y) -> {
    MutList.add!(key, duplicates);
    x + y
};
let m3 = DelayMap.unionWithKey(merge, m1, m2)
```

The merge operation is impure and hence unionWith is evaluated eagerly. This ensures that the mutable list duplicates is updated correctly.

4.5 Summary

We have demonstrated the usefulness of reifyEff by using it in four data structures:

- We refactored the Flix Standard Library implementation of the Set and Map data structures to use selective parallel evaluation.
- We have introduced two new data structures: DelayList which uses selective lazy evaluation and DelayMap which uses selective lazy and parallel evaluation.

Table 1 shows an overview of the data structures that we have implemented. The Data Structure column gives the name of the data structure. The Lines column gives the number of Flix source code lines (excluding tests). The Tests column gives the number of manually written unit tests. The Functions column gives the number of functions implemented on the data structure. Most functions are first-order and "terminal". For example, Set.memberOf is first-order and terminal, i.e., it does not transform the Set but rather returns a value. The `@LazyWhenPure` gives the number of functions that use purity reflection to enable lazy evaluation. The `@ParallelWhenPure` gives the number of functions that use purity reflection to enable parallel evaluation. The LWP + PWP gives the number of functions that use purity reflection to enable *both* lazy and parallel evaluation. For example, the line for DelayMap shows that the data structure is implemented in 786 lines of Flix code with 298 unit tests. The data structure offers 58 functions of which 10 use purity reflection to enable lazy evaluation, 7 use purity reflection to enable parallel evaluation, and 2 use purity reflection to enable both. Except for DelayList, these three data structures build on a Red-Black Tree, whose line counts are not included.

5 Compilation Strategies

We now discuss two ways to implement purity reflection: one based on runtime dispatch and the other on a novel form of effect-aware monomorphization. Both approaches rely on the type and effect system since we cannot readily determine if an expression will be pure at runtime. In other words, our extension of Flix only assigns meaning to well-typed programs.

5.1 Runtime Dispatch

Given a well-typed program, we can annotate each closure with a Boolean (or Boolean formula) to track if it is pure or impure. For a first-order function (i.e., a function without function arguments), its purity is readily determined by the typing judgment. For a higher-order function, its purity may depend on its function arguments. In this case, we label the closure with a Boolean formula that refers to the purity of the closure arguments. Given such an annotated closure, the reifyEff construct can be implemented by simply inspecting these annotations (and potentially evaluating a Boolean formula). For example, if we have the program fragment:

```
let f = x -> x + 1;
let g = x -> println(x);
let h = u -> v -> x -> u(v(x));
```

Then the closure of f stores a Boolean formula which is the constant true, the closure of g stores a Boolean formula which is the constant false, and the closure of h stores a Boolean formula which is the conjunction of the two bits of the higher-order arguments u and v. Thus, at runtime, the purity of h can be computed once u and v are known.

The cost of the runtime dispatch strategy is that we must store a Boolean formula with each closure. For first-order functions, this is just the constant true or false. For higher-order functions, it is a formula with several variables corresponding to its higher-order functions. Thus, in general, these formulas will be small since most functions are first-order and since higher-order functions tend to have only a few function arguments. Hence, the increase in code size should be modest. At runtime, the reifyEff construct has to evaluate small Boolean formulas which should be fast.

5.2 Effect-Aware Monomorphization

As an alternative to runtime dispatch, we propose an effect-aware form of monomorphization.

Monomorphization is a compile-time transformation that replaces polymorphic functions with copies that are specialized to the concrete types of their arguments. For example, if List.map is used with both integer and string lists, then monomorphization generates two copies of List.map: one specialized to integers and one specialized to strings. Monomorphization avoids boxing at the cost of larger executables. In practice, code size can be significantly reduced with the proper use of inlining and dead code elimination. A potential downside of monomorphization is that it may prevent separate compilation.

Before our work, the Flix compiler performed specialization of type variables and erased effect variables. In this work, we have extended the Flix compiler to specialize effect variables. In other words, the Flix compiler is now able to generate two versions of List.map[Int]: one specialized for pure functions and one specialized for impure functions. The upshot is that reifyEff can be eliminated – entirely at compile-time – because its argument is statically known to be pure or impure. Thus, the use of reifyEff incurs zero runtime overhead.

A technical detail is that during monomorphization a type or effect variable can potentially be left un-instantiated. For example, in the expression `Nil == Nil` each `Nil` can be given any type α. For effect variables, we can use two strategies to deal with such situations:

- (**Opportunistic**) We opportunistically treat all un-instantiated effect variables as pure. This is sound because the type system is closed under substitution (i.e., if a variable is free we may substitute it by `T`).
- (**Conservative**) We reject programs that contain un-instantiated effect variables during monomorphization. The programmer can always resolve the situation with a type (or effect) annotation.

In our extension of Flix, we choose the conservative option because it is consistent with how ordinary un-instantiated type variables are treated. As an example, the following (contrived) program has an un-instantiated effect variable:

```
let f = g -> reifyEff(g) {
    case Pure(w) => g
    case _       => g
};
f(f)
```

Here the type of f is: $\forall \alpha_1, \alpha_2, \beta.\, (\alpha_1 \xrightarrow{\beta} \alpha_2) \xrightarrow{\mathtt{T}} (\alpha_1 \xrightarrow{\beta} \alpha_2)$. When f is applied to *itself* it returns a function of the same type which has an un-instantiated effect variable even after monomorphization. In Section 7, we investigate how common un-instantiated effect variables are in real programs. We have not observed un-instantiated effect variables in existing code.

5.3 Discussion

We believe that both the runtime dispatch and the effect-aware monomorphization approaches are viable. We decided to implement purity reflection via monomorphization since:

- Flix already uses monomorphization to eliminate parametric polymorphism.
- Monomorphization enables more aggressive optimizations performed by the Flix inliner.
- Monomorphization ensures that the technique imposes zero runtime overhead.

Finally, as our experiments in Section 7 demonstrate, the increase in compilation time and code size is small.

6 Implementation

We have implemented purity reflection as an extension of the Flix programming language.

6.1 The Flix Programming Language

Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, polymorphic effects, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [24, 25, 26, 27, 28].

The Flix compiler project, including the standard library and tests, consists of 230,000 lines of Flix and Scala code. Adding reifyEff and effect-aware monomorphization required less than 2,000 lines of code. The extended Flix Standard Library required approximately 4,000 lines of code with unit tests (see Section 4).

Flix, with our extension, is open source, ready for use, and freely available at:

`https://flix.dev/` and `https://github.com/flix/flix/`

6.2 Integration with Type Classes

Flix supports type classes and higher-kinded types. The Flix compiler implements type classes using monomorphization, i.e., there is no dynamic dispatch or dictionary passing. This also has the advantage that purity reflection works with type classes without any modifications. For example, if a polymorphic function requires a Foldable instance because of a call to Foldable.count then during monomorphization the call to Foldable.count will be replaced by a call to the appropriate instance function. Thus, the benefits of selective lazy and/or parallel evaluation are available even for polymorphic functions that use type classes.

7 Evaluation

We now turn to the question of the viability of an implementation strategy based on effect-aware monomorphization. In particular, we consider the following research questions:

- **RQ1**: What is the impact of effect-aware monomorphization on compilation time?
- **RQ2**: What is the impact of effect-aware monomorphization on code size?
- **RQ3**: How common are un-instantiated effect variables in practice?

7.1 RQ1 and RQ2: Impact on compilation time and code size

Monomorphization specializes (i.e., copies) functions for each of their concrete type arguments. For example, the List.map function has the polymorphic type:

$$\forall a, \forall b, \forall e \,.\, (a \xrightarrow{e} b) \to \mathsf{List}[a] \xrightarrow{e} \mathsf{List}[b]$$

Under standard monomorphization, the List.map function is copied for every instantiation of the type variables a and b. If, for example, List.map is called with the two functions that have types: Int32 → Bool and String → Int32, we would get two copies of List.map.

With effect-aware monomorphization, a function is copied for every instantiation of its type variables *and* its effect variables. In other words, List.map will be copied for every instantiation of a, b, and e. Every effect variable is either pure or impure which means that we can get two copies of a function per effect variable (in addition to its other type variables). In the worst case, this can lead to an exponential blow-up in the number of copies.

We can construct a worst-case example with three effect variables:

```
def hof(f: a -> b & ef1, g: a -> b & ef2, h: a -> b & ef3): Unit = ...

def p(): Unit & Pure = ...    // a pure function
def i(): Unit & Impure = ...  // an impure function

def main(): Unit & Impure =
    hof(p, p, p);
    hof(i, p, p);
    // ... omitted for brevity ...
    hof(i, i, i)
```

The higher order function hof takes three function arguments f, g, and h. Each function argument has an effect variable, which can be instantiated to pure or impure. Inside main, we call hof with all possible combinations of pure and impure function arguments. There are $2^3 = 8$ combinations of these. This means that during effect-aware monomorphization we will construct 8 copies of hof, which duplicates its entire function body. If hof is large this can lead to a blow-up in compilation time and code size.

Table 2 Impact of Effect-Aware Monomorphization on Compilation Time and Code Size.
†: The "DeliveryDate" and "Stratifier" programs depend on the Flix Datalog engine, which is a part of the Flix Standard Library and implemented in Flix itself. Hence, these programs are not actually 35-116 lines of code, but more accurately thought of as 35-116 lines of code *plus* the 3,055 lines of code used to implement the Datalog engine in Flix.

		Std. Monomorphization			Effect-Aware Monomorphization		
Program	Lines	Time	Bytes	Classes	Time	Bytes	Classes
Standard Library	33,689	4.6s	4,954	8	–	–	–
BoolTable	206	4.6s	766,638	759	4.6s (+0%)	766,546 (+0%)	759 (+0%)
DeliveryDate †	35	5.1s	2,913,210	2,762	5.1s (+0%)	3,019,959 (+4%)	2,888 (+5%)
fcwg	2,796	5.0s	1,700,982	1,911	5.0s (+0%)	1,699,107 (+0%)	1,911 (+0%)
flixball	1,767	4.8s	875,858	1,012	4.8s (+0%)	877,762 (+0%)	1,012 (+0%)
IfNoSub	1,870	5.2s	1,967,390	1,849	5.2s (+0%)	2,046,480 (+4%)	1,921 (+4%)
JSON	348	4.9s	486,521	573	4.9s (+0%)	487,025 (+0%)	573 (+0%)
Regex	1,891	4.5s	223,429	316	4.5s (+0%)	222,479 (+0%)	316 (+0%)
Stratifier †	116	4.9s	3,063,802	2,925	4.9s (+0%)	3,191,718 (+4%)	3,070 (+5%)
TestDelayList	3,060	5.1s	4,050,485	5,304	5.1s (+0%)	4,174,417 (+3%)	5,431 (+2%)
TestDelayMap	2,039	5.1s	4,111,046	5,090	5.1s (+0%)	4,750,187 (+16%)	5,631 (+11%)
TestMap	2,780	5.4s	5,230,078	5,782	5.4s (+0%)	5,572,129 (+7%)	6,107 (+6%)
TestSet	1,640	4.9s	2,598,250	2,759	4.9s (+0%)	2,757,388 (+6%)	2,930 (+6%)

Analysis

Given a polymorphic function f with n type parameters (quantified type variables) and m effect parameters (quantified Boolean variables), effect-aware monomorphization will create at most $t^n \times 2^m$ copies of f where t is the number of types that occur in the program after type checking but before monomorphization. In practice, type-based monomorphization does not lead to an exponential blow-up, but what about effect-aware monomorphization?

Table 2 shows the impact of effect-aware monomorphization on compilation time and code size for several Flix programs. We briefly discuss each program: The Flix "Standard Library" is included for completeness. When the library is compiled alone, without any entry point, a mere 8 Java classes are generated. These 8 classes are hard-coded and are always emitted. One class represents the Unit value. Other classes represent various exceptions. "BoolTable" is a Flix program to generate a table of smallest formulas for all Boolean functions of 4 arguments. "DeliveryDate" is a Flix program that uses first-class Datalog constraints with lattice semantics to compute the earliest delivery date for a "component" that consists of subcomponents, each with its delivery date and assembly time. "fcwg" is a Flix program generator that generates wrapper code for Java classes. "flixball" is a basic multi-player, 2-dimensional shooter game, run in the console. Bots compete in a last-player standing arena, taking simultaneous turns to rotate, move, or fire their weapon. "IfNoSub" is an implementation of Algorithm W for Flix written in Flix. It captures the relational nullable type system from [28]. "JSON" is, as the name implies, a JSON library for Flix. "Regex" is, as the name implies, a Regex library for Flix (based on Java's regex package). "Stratifier" is a Flix program that uses first-class Datalog constraints with lattice semantics to implement a version of Ullman's algorithm to compute the stratification of a Datalog program. "TestX" is the collection of unit tests for the data structure X.

We now explain each column of the table: The Program column gives the name of the Flix program. The Lines column shows the number of lines of code in the program (excluding the Flix Standard Library), The Time column shows the total compilation time in seconds (without effect-aware monomorphization). The Bytes column shows the number of bytes generated by the compiler (without effect-aware monomorphization). The Classes column shows the number of classes generated by the Flix compiler (without effect-aware monomorphization). The last three columns then repeat but now with effect-aware monomorphization. The numbers in parentheses show the percentage increase (resp. decrease) in the specific measurement.

For example, the "IfNoSub" program consists of $1,870$ lines of code (in addition to the 30,000 lines of code from the Flix Standard Library). Under type-based monomorphization, the Flix compiler generates $1,849$ Java classes, totaling $1,967,390$ bytes in $5,2s$. With effect-aware monomorphization, the compiler generates $1,921$ Java classes totaling $2,046,480$ bytes in the same amount of time. This represents a 4% increase in code and classes.

As a second example, the "TestDelayMap" program consists of $2,039$ lines of code, which contain all the unit tests for the DelayMap data structure. Every purity reflective function is tested with both a pure and impure function argument. Under type-based monomorphization, the Flix compiler generates $5,090$ Java classes totaling $4,111,046$ bytes in $5,1s$. With effect-aware monomorphization, the compiler generates $5,631$ Java classes totaling $4,750,187$ bytes within the same time. This represents an 16% increase in code and an 11% increase in classes.

As Table 2 shows, the cost of effect-aware monomorphization is low. For real programs, there is no increase in compile time and the increase in code size is between 0% to 5%.
We offer a few explanations why real programs show only a modest increase:

- Most functions are first-order (i.e., they do not take function arguments). A first-order function cannot be copied by effect-aware monomorphization, hence it cannot lead to increased compilation time or code size. To give two examples: In the Set module 17/45 functions are higher-order whereas in the String module 24/94 functions are higher-order.
- The majority of function calls are to first-order functions. For example, List.sum is probably more widely used than e.g., List.zipWith3. In other words, there are fewer higher-order functions than first-order functions, and they are on the balance also less likely to be used.
- When a higher-order function is used, it is not necessarily used with both pure and impure function arguments. For example, in many of the programs, higher-order functions are always used with a pure or an impure function argument, but more rarely with both.

In sum, we conclude that effect-aware monomorphization is a viable implementation strategy.

7.2 RQ3: How common are un-instantiated effect variables in practice?

As discussed in Section 5.2, an effect variable is potentially left un-instantiated during monomorphization. We showed a carefully crafted example – which relied on self-application – that would lead to an un-instantiated effect variable. We discussed two sound solutions: (i) an optimistic strategy that treats every un-instantiated variable as true (i.e., as pure), and (ii) a conservative strategy that rejects programs with un-instantiated (effect) variables. Flix uses the conservative strategy.

An important empirical question is then how common such situations are. In more than 100,000 lines of Flix code, we have never encountered a single un-instantiated effect. In fact, we have only been able to trigger the rejection with our carefully crafted example. We conclude that un-instantiated effect variables are not of practical concern. Intuitively, most expressions are either pure or impure. A few expressions are effect-polymorphic, but they are almost always called with pure or impure function arguments. Consequently, during monomorphization, we always end up with expressions that are either pure or impure.

8 Related Work

We consider related work along three axes: type and effect systems, reflection, and streams.

8.1 Type and Effect Systems

The Flix type and effect system is based on Hindley-Milner [12, 30, 7] extended with Boolean unification [29, 27]. The Flix system is effect-polymorphic, a notion that goes back to Lucassen et al. [23].

Algebraic effects is a hot research topic [33, 14, 19, 1, 21, 3, 4]. An algebraic effect system allows the programmer to define a collection of effects that can be invoked and interpreted by *effect handlers* installed on the stack (similar to exceptions). A type and effect system for an algebraic effect system ensures that all effects are ultimately handled. Most prototype programming languages that support algebraic effects *and* complete type inference are based on row polymorphism. Purity reflection seems orthogonal to algebraic effects; we are not interested in a collection of effects nor in how to interpret them. We are interested in enabling higher-order functions to selectively use lazy or parallel evaluation when passed pure function arguments. As interesting future work, we can imagine a type and effect system that tries to combine algebraic effects with purity reflection while retaining complete type inference.

A line of research has used uniqueness and ownership type systems to prevent data races and deadlocks and to enable parallelism [2, 6, 9]. Boyapati et al. present a type system that prevents data races and deadlocks. In the system, programmers partition locks into equivalence classes and define a partial order on them. The type checker then verifies that the locks are acquired in descending order [2]. Craik and Kelly present a type, effect, and ownership system that uses read-and-write effect sets to reason about data dependence. This information is then exploited to enable data or task parallelism [6]. Gordon et al. present a type and effect system that restricts updates to mutable memory shared by multiple threads. The system relies on a combination of immutable and uniqueness types, which ensure the absence of data races [9].

8.2 Type Case and Effect Reflection

Tarditi et al. propose *type case*, a meta-programming construct that enables polymorphic functions to reflect on their concrete type arguments [11, 36]. In the TIL Standard ML compiler, type casing is used to implement several polymorphic functions more efficiently. For example, an array indexing ("subscript") operation can be implemented more efficiently if the compiler knows the concrete type of the underlying array. One might think of our work as an *effect case* which is used to enable selective lazy or parallel evaluation.

Long et al. propose a calculus and type system with reflection for effects representing region accesses [22]. Their effects are first-class expressions that can be inspected by pattern matching. The features of their system are orthogonal to our system: they have a hybrid approach, based on static and dynamic types; their calculus provides over-approximating (may) and under-approximating (must) types, and is based on sub-typing/effecting and refinement types. Instead, we support type and effect polymorphism with inference based on Boolean unification. We also provide an implementation in a programming language.

8.3 Streams

Broadly speaking, the relation between our work and work on streams is that most stream implementations aim to provide lazy and/or parallel evaluation capabilities in programming languages that are eager and impure, at the risk of unsoundness. Purity reflection allows library authors to soundly determine *when* it is safe to use lazy and/or parallel evaluation.

Wadler et al. introduce the notion of "odd" and "even" lazy lists [38]. Wadler et al.'s observation is that in the standard definition of a lazy list, the first element of the list is eager whereas the rest of the list is lazy. In other words, the lazy list is not entirely lazy which can lead to unexpected behavior. We agree with this observation and in our real implementation of DelayList every element is lazy.

Jones et al. present an extension of Haskell, where the programmer can define a collection of rewrite rules that are applied to the program by the Haskell compiler [13]. Using rewrite rules, a programmer (typically a library author) can express program optimizations as a collection of transformations. The mechanism is extended and applied to stream fusion in [5]. The rewrite mechanism could provide an alternative implementation of purity reflection. Similar to our solution using monomorphization, rewrite rules are applied at compile time and induce no run-time overhead. However, we see two advantages in our solution based on purity reflection: First, rewrite rules only apply if there is a syntactic pattern match, while our technique also applies across various let-bindings or even function calls, since it is based on a type and effect system that propagates information. Second, a programmer could easily add unsound rewrite rules, while our type system provides some guarantees; in particular, impure functions cannot be postponed.

Prokopec et al. show that the overhead of functional combinators (e.g., filter, map) on the JVM can be overcome with a sufficiently aggressive JIT compiler [34]. Kiselyov et al. present a technique to overcome the overhead of stream operations through staging [18]. Møller and Veileborg propose to use static analysis to eliminate the overhead of stream pipelines in Java [32]. In summary, the bulk of this work has focused on *how* to make streams execute faster. In contrast, our work concerns *when* it is safe to do so. We use monomorphization to implement purity reflection. After monomorphization, we have a mono-typed program where each use of a non-terminal operation (e.g., filter and map) has been replaced by its eager or lazy version. We can pass the monomorphed AST to any technique that performs stream fusion without the risk of unsoundness.

Khatchadourian et al. present a study on the use and misuse of streams in Java [17]. Interestingly two of their findings are: "Finding 1: Stream parallelization is not widely used" and "Finding 3: Streams tend not to have side effects." The former finding could suggest that even though parallel streams are readily available, developers are either unaware or reluctant to use them. With purity reflection, the choice of whether to use parallelism rests not just with the programmer but also with the library author. This suggests that purity reflection can help programmers by exploiting parallelism when they did not consider it themselves.

Khatchadourian et al. also present an automatic technique to refactor code to use streams more efficiently [16]. The technique is based on heavy-weight program analysis. Flix, with purity reflection, offers an alternative approach where reasoning about and reflecting on purity is built directly into the language.

9 Conclusion

We have proposed *purity reflection*, a new programming language feature that enables higher-order functions to vary their behavior depending on the purity of their function arguments. Purity reflection enables selective use of lazy and/or parallel evaluation, while ensuring that side effects are never lost or re-ordered. We have implemented purity reflection in the Flix programming language. We have retrofitted and extended the Flix Standard Library with new data structures that automatically use lazy or parallel evaluation when it is safe to do so. Effect-aware monomorphization provides a mechanism to implement purity reflection as

a construct that is entirely eliminated at compile-time. Therefore, the technique imposes no run-time overhead. Experimental results show that the cost of effect-aware monomorphization in compilation time and code size is minimal.

References

1 Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1), 2015. doi:10.1016/j.jlamp.2014.02.001.

2 Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2002. doi:10.1145/582419.582440.

3 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effect Handlers for the Masses. *Proc. of the ACM on Programming Languages*, 2(OOPSLA), 2018. doi:10.1145/3276481.

4 Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. Effects as Capabilities: Effect Handlers and Lightweight Effect Polymorphism. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428194.

5 Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream Fusion: From Lists to Streams to Nothing at All. *International Conference of Functional Programming (ICFP)*, 42(9), 2007. doi:10.1145/1291220.1291199.

6 Andrew Craik and Wayne Kelly. Using Ownership to Reason about Inherent Parallelism in Object-Oriented Programs. In *International Conference on Compiler Construction (CC)*, 2010. doi:10.1007/978-3-642-11970-5_9.

7 Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, The University of Edinburgh, 1984.

8 Jacques Garrigue. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*, 2004. doi:10.1007/978-3-540-24754-8_15.

9 Colin S Gordon, Matthew J Parkinson, Jared Parsons, Aleks Bromfield, and Joe Duffy. Uniqueness and Reference Immutability for Safe Parallelism. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2012. doi:10.1145/2398857.2384619.

10 Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based Memory Management in Cyclone. In *Programming Language Design and Implementation (PLDI)*, 2002. doi:10.1145/512529.512563.

11 Robert Harper and Greg Morrisett. Compiling Polymorphism using Intensional Type Analysis. In *Principles of Programming Languages (POPL)*, 1995. doi:10.1145/199448.199475.

12 Roger Hindley. The Principal Type-scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society (AMS)*, 1969.

13 Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell Workshop*, 2001.

14 Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in Action. *International Conference on Functional Programming (ICFP)*, 48(9), 2013. doi:10.1145/2544174.2500590.

15 Ohad Kammar and Gordon D. Plotkin. Algebraic Foundations for Effect-dependent Optimisations. In *Principles of Programming Languages (POPL)*, 2012. doi:10.1145/2103656.2103698.

16 Raffi Khatchadourian, Yiming Tang, and Mehdi Bagherzadeh. Safe Automated Refactoring for Intelligent Parallelization of Java 8 Streams. *Science of Computer Programming*, 2020. doi:10.1016/j.scico.2020.102476.

17 Raffi Khatchadourian, Yiming Tang, Mehdi Bagherzadeh, and Baishakhi Ray. An Empirical Study on the Use and Misuse of Java 8 Streams. In *Fundamental Approaches to Software Engineering (FASE)*, 2020. doi:10.1007/978-3-030-45234-6_5.

18 Oleg Kiselyov, Aggelos Biboudis, Nick Palladinos, and Yannis Smaragdakis. Stream Fusion, to Completeness. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009880.

19 Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible Effects: An Alternative to Monad Transformers. *Haskell Workshop*, 2013. doi:10.1145/2578854.2503791.

20 Daan Leijen. Extensible Records with Scoped Labels. *Trends in Functional Programming (TFP)*, 2005.

21 Daan Leijen. Type Directed Compilation of Row-typed Algebraic Effects. In *Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009872.

22 Yuheng Long, Yu David Liu, and Hridesh Rajan. First-class Effect Reflection for Effect-Guided Programming. In *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016. doi:10.1145/3022671.2984037.

23 John M Lucassen and David K Gifford. Polymorphic Effect Systems. In *Principles of Programming Languages (POPL)*, 1988.

24 Magnus Madsen. The Principles of the Flix Programming Language. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2022. doi:10.1145/3563835.3567661.

25 Magnus Madsen and Ondřej Lhoták. Fixpoints for the Masses: Programming with First-class Datalog Constraints. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428193.

26 Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. Flix: A Meta Programming Language for Datalog. In *Proc. International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog-2.0 2022)*, 2022.

27 Magnus Madsen and Jaco van de Pol. Polymorphic Types and Effects with Boolean Unification. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428222.

28 Magnus Madsen and Jaco van de Pol. Relational Nullable Types with Boolean Unification. *Proc. of the ACM on Programming Languages*, 5(OOPSLA), 2021. doi:10.1145/3485487.

29 Urusula Martin and Tobias Nipkow. Boolean Unification - The Story So Far. *Journal of Symbolic Computation*, 1989.

30 Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 1978.

31 Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml: Functional Programming for the Masses*. O'Reilly Media, 2013.

32 Anders Møller and Oskar Haarklou Veileborg. Eliminating Abstraction Overhead of Java Stream Pipelines using Ahead-of-Time Program Optimization. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428236.

33 Matija Pretnar and Gordon D Plotkin. Handling Algebraic Effects. *Logical Methods in Computer Science*, 2013. doi:10.2168/LMCS-9(4:23)2013.

34 Aleksandar Prokopec, David Leopoldseder, Gilles Duboscq, and Thomas Würthinger. Making Collection Operations Optimal with Aggressive JIT Compilation. In *Proc. International Symposium on Scala*, 2017. doi:10.1145/3136000.3136002.

35 John C Reynolds. The meaning of types from intrinsic to extrinsic semantics. *BRICS Report Series*, 7(32), 2000.

36 David Tarditi, Greg Morrisett, Perry Cheng, Chris Stone, Robert Harper, and Peter Lee. TIL: A Type-directed Optimizing Compiler for ML. In *Programming Language Design and Implementation (PLDI)*, 1996. doi:10.1145/249069.231414.

37 Mads Tofte and Jean-Pierre Talpin. Region-based Memory Management. *Information and Computation*, 1997. doi:10.1006/inco.1996.2613.

38 Philip Wadler, Walid Taha, and David MacQueen. How to Add Laziness to a Strict Language Without Even Being Odd. In *SML'98, The SML workshop*, 1998.

39 Mitchell Wand. Complete Type Inference for Simple Objects. In *Logic in Computer Science*, 1987.

40 Keith Wansbrough and Simon L. Peyton Jones. Once Upon a Polymorphic Type. In *Principles of Programming Languages (POPL)*, 1999. doi:10.1145/292540.292545.