STATE OF EFF

Matija Pretnar
University of Ljubljana, Slovenia

2010

EFF 1.0

← How eff handles built-in effects

Programming with effects I: Theory →

Programming with effects II: Introducing eff

27 September 2010 Matija Pretnar Computation, Eff, Guest post, Programming, Software, Tutorial

[UPDATE 2012-03-08: since this post was written eff has changed considerably. For updated information, please visit the eff page.]

**This is a second post about the programming language eff. We covered the theory behind it in a previous post. Now we turn to the programming language itself.

Please bear in mind that eff is an academic experiment. It is not meant to take over the world. Yet. We just wanted to show that the theoretical ideas about the algebraic nature of computational effects can be put into practice. Eff has many superficial similarities with Haskell. This is no surprise because there is a precise connection between algebras and monads. The main advantage of eff over Haskell is supposed to be the ease with which computational effects can be combined.

Installation

If you have Mercurial installed (type `hg` at command prompt to find out) you can get eff like this:

```
$ hg clone http://hg.andrej.com/eff/ eff
```

Otherwise, you may also download the latest source as a `.zip` or `.tar.gz`, or visit the repository with your browser for other versions. Eff is released under the simplified BSD License.

To compile eff you need Ocaml 3.11 or newer (there is an incompatibility with 3.10 in the Lexer module), ocamlbuild, and Menhir (which are both likely to be bundled with Ocaml). Put the source in a suitable directory and compile it with `make` to create the Ocaml bytecode executable `eff.byte`. When you run it you get an interactive shell without line editing capabilities. If you never make any typos that should be fine, otherwise use one of the line editing wrappers, such as rlwrap or ledit. A handy shortcut `eff` runs `eff.byte` wrapped in rlwrap.

Syntax

Mathematics and Computation

A blog about mathematics for computers

← How eff hand

Programming with effects I: Theory →

Progr

27 Septer

```
effect state x:
    operation lookup ():
        (lambda s: yield s s)
    operation update s_new:
        (lambda s: yield () s_new)
return y:
    (lambda s: (s, y))
finally f: f x
```

[UPDATE 2

**This is a ning language itself.

Please be l ideas about the algebraic nature of , a precise connection between combined.

Installation

If you have Mercurial installed (type `hg` at command prompt to find out) you can get eff like t

```
$ hg clone http://hg.andrej.com/eff/ eff
```

Otherwise, you may also download the latest source as a `.zip` or `.tar.gz`, or visit the repository with your browser for other versions. Eff is released under the simplified BSD License.

To compile eff you need Ocaml 3.11 or newer (there is an incompatibility with 3.10 in the Lexer module), ocamlbuild, and Menhir (which are both likely to be bundled with Ocaml). Put the source in a suitable directory and compile it with `make` to create the Ocaml bytecode executable `eff.byte`. When you run it you get an interactive shell without line editing capabilities. If you never make any typos that should be fine, otherwise use one of the line editing wrappers, such as rlwrap or ledit. A handy shortcut `eff` runs `eff.byte` wrapped in rlwrap.

Syntax

2011

EFF 2.0

Programming with Algebraic Effects and Handlers

Andrej Bauer[a], Matija Pretnar[a]

[a]*Faculty of Mathematics and Physics, University of Ljubljana, Slovenia*

Abstract

Eff is a programming language based on the algebraic approach to computational effects, in which effects are viewed as algebraic operations and effect handlers as homomorphisms from free algebras. *Eff* supports first-class effects and handlers through which we may easily define new computational effects, seamlessly combine existing ones, and handle them in novel ways. We give a denotational semantics of *Eff* and discuss a prototype implementation based on it. Through examples we demonstrate how the standard effects are treated in *Eff*, and how *Eff* supports programming techniques that use various forms of delimited continuations, such as backtracking, breadth-first search, selection functionals, cooperative multi-threading, and others.

Introduction

Eff is a programming language based on the algebraic approach to effects, in which computational effects are modelled as operations of a suitably chosen algebraic theory [12]. Common computational effects such as input, output, state, exceptions, and non-determinism, are of this kind. Continuations are not algebraic [4], but they turn out to be naturally supported by *Eff* as well. Effect handlers are a related notion [14, 19] which encompasses exception handlers, stream redirection, transactions, backtracking, and many others. These are modelled as homomorphisms induced by the universal property of free algebras.

Each algebraic theory gives rise to a monad [1, 11], although the operations cannot be reconstructed from it. Algebraic theories have their own virtues, though. Effects are combined more easily than monads [5], and the interaction between effects and handlers offers new ways of programming. An experiment in the design of a programming language based on the algebraic approach therefore seems warranted.

Philip Wadler once opined [21] that monads as a programming concept would not have been discovered without their category-theoretic counterparts, but once they were, programmers could live in blissful ignorance of their origin. Because the same holds for algebraic effects and handlers, we streamlined the paper for the benefit of programmers, trusting that connoisseurs will recognize the connections with the underlying mathematical theory.

Email addresses: andrej@andrej.com (Andrej Bauer), matija@pretnar.info (Matija Pretnar)

```
type 'a ref = effect
  operation lookup: unit -> 'a
  operation update: 'a -> unit
end

let state r x = handler
  | r#lookup () k -> (fun s -> k s s)
  | r#update s' k -> (fun s -> k () s')
  | val y -> (fun s -> (y, s))
  | finally f -> f x
```

Programming with Algebraic Effects and Handlers

Andrej Bauer[a]

Email addresses: andrej@andrej.com

Preprint submitted to Elsevier *November 12, 2013*

2012

EFF 3.0 ?

2013

EFFECT SYSTEM

Logical Methods in Computer Science
Vol. 10(3:21)2014, pp. 1–43
www.lmcs-online.org

Submitted Dec. 5, 2013
Published Sep. 12, 2014

INFERRING ALGEBRAIC EFFECTS

MATIJA PRETNAR

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
e-mail address: matija@pretnar.info

Abstract. We present a complete polymorphic effect inference algorithm for an ML-style language with handlers of not only exceptions, but of any other algebraic effect such as input & output, mutable references and many others.

Our main aim is to offer the programmer a useful insight into the effectful behaviour of programs. Handlers help here by cutting down possible effects and the resulting lengthy output that often plagues precise effect systems. Additionally, we present a set of methods that further simplify the displayed types, some even by deliberately hiding inferred information from the programmer.

Though Haskell [10] fans may not think it is better to write impure programs in ML [18], they do agree it is easier. You can insert a harmless printout without rewriting the rest of the program, and you can combine multiple effects without a monad transformer. This flexibility comes at a cost, though — ML types offer no insight into what effects may happen. The suggested solution is to use an effect system [16, 29, 4, 31, 33, 3, 27], which enriches existing types with information about effects.

An effect system can play two roles: it can be *descriptive* and inform about potential effects, and it can be *prescriptive* and limit the allowed ones. In this paper, we focus on the former. It turns out that striking a balance between expressiveness and simplicity of a descriptive effect system is hard. One of the bigger problems is that effects tend to pile up, and if the effect system takes them all into account, we are often left with a lengthy output listing every single effect there is.

In this paper, we present a complete inference algorithm for an expressive and simple descriptive polymorphic effect system of *Eff* [2] (freely available at `http://eff-lang.org`), an ML-style language with handlers of not only exceptions, but of any other *algebraic effect* [22] such as input & output, non-determinism, mutable references and many others [23, 2]. Handlers prove to be extremely versatile and can express stream redirection, transactional memory, backtracking, cooperative multi-threading, delimited continuations, and, like monads, give programmers a way to define their own. And as handlers eliminate effects, they make the effect system *non-monotone*, which helps with the above issue of a snowballing output.

2012 ACM CCS: [**Theory of computation**]: Semantics and reasoning—Program reasoning—Program analysis.

Key words and phrases: algebraic effects, effect handlers, effect inference, effect system.

LOGICAL METHODS IN COMPUTER SCIENCE

DOI:10.2168/LMCS-10(3:21)2014

2015-2016

EFFICIENT COMPILATION

```
effect Put: int -> unit
effect Get: unit -> int

let rec loop n =
  if n = 0 then () else
    perform (Put (perform (Get ()) + 1));
    loop (n - 1)

let state_handler = handler
  | effect (Put s') k -> (fun _ -> k () s')
  | effect (Get ()) k -> (fun s -> k s s)
  | _ -> (fun s -> s)
```

```
let main n =
  (with state_handler handle loop n) 0
```

```
let main n =
  let rec state_handler_loop m s =
    if m = 0 then s
             else state_handler_loop (m - 1) (s + 1)
  in
  state_handler_loop n 0
```

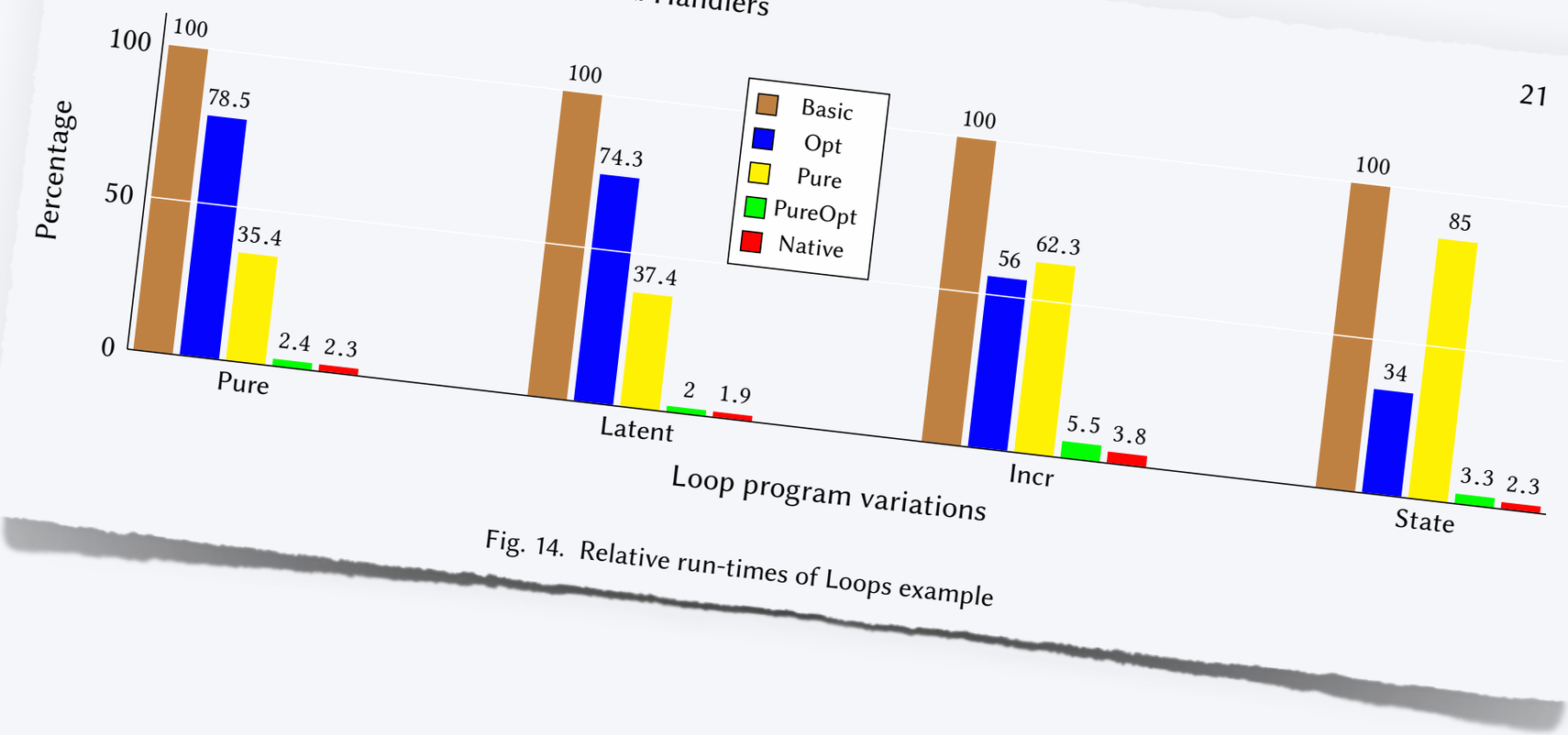


Fig. 14. Relative run-times of Loops example

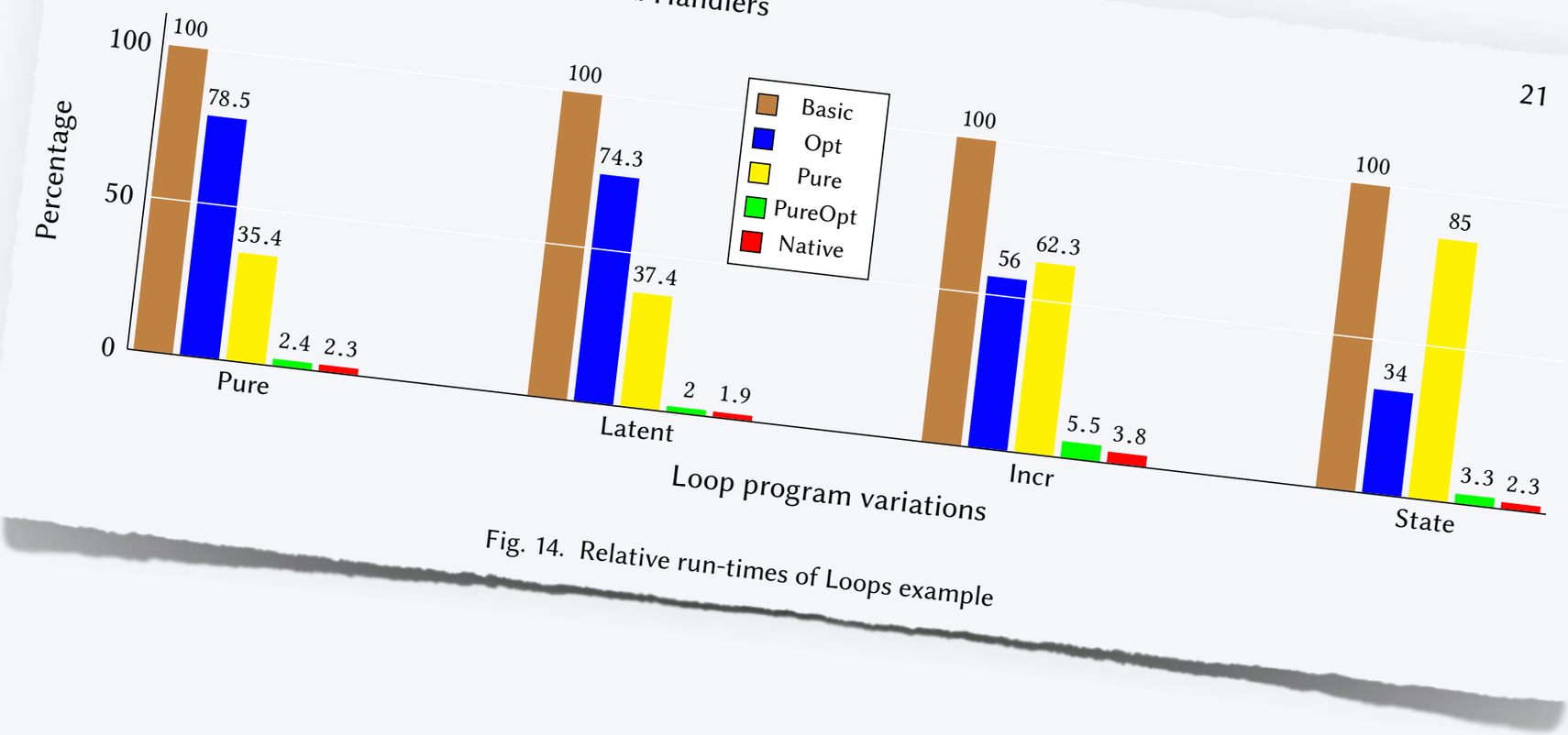
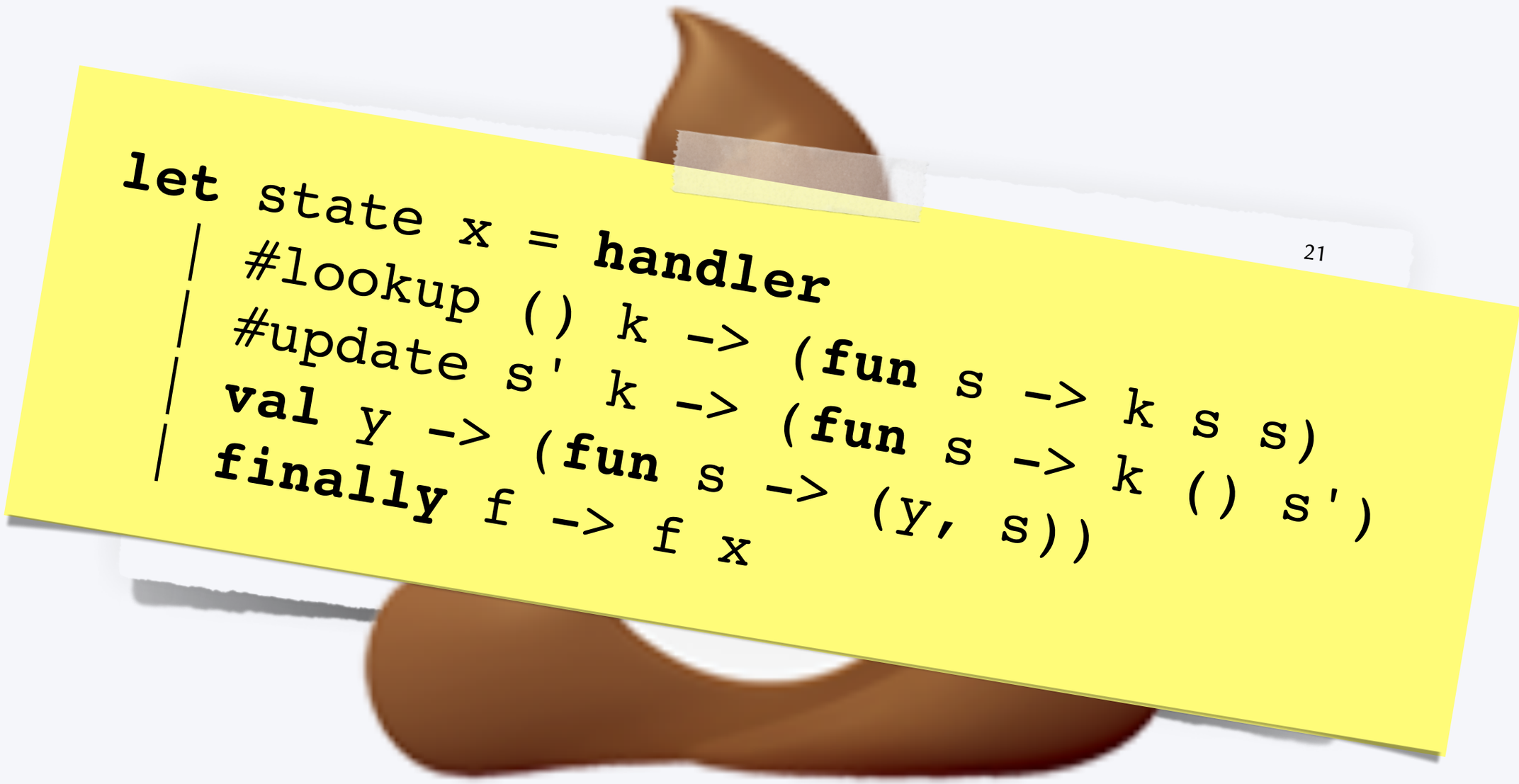


Fig. 14. Relative run-times of Loops example


Efficient Compilation of Algebraic Effects and H
Percentage
100
50
0
100
78.5
35.4
2.4
2.3
Pure
21
100
85
34
3.3
2.3
State


Efficient Compilation of Algebraic Effects and H
Percentage
100
50
0
100
78.5
35.4
2.4
2.3
Pure
21
100
85
34
3.3
2.3
State



```
let state x = handler
  | #lookup () k -> (fun s -> k s s)
  | #update s' k -> (fun s -> k () s')
  | val y -> (fun s -> (y, s))
  | finally f -> f x
```

2017-2019

EXPLICIT SUBTYPING

```
let apply_if p f x =
  if p x then
    f x
  else
    x
```

$$
\begin{array}{l}
\texttt{fun}\ p \mapsto \texttt{return}\ (\texttt{fun}\ f \mapsto \texttt{return}\ (\texttt{fun}\ x \mapsto (\\
\quad \texttt{do}\ b \leftarrow p\, x; \\
\quad\quad \texttt{if}\ b\ \texttt{then}\ f\, x\ \texttt{else}\ \texttt{return}\ x \\
)))
\end{array}
$$

$$
\begin{array}{l}
\texttt{fun}\ (p : \alpha_1 \to \texttt{bool}) \mapsto \texttt{return}\ (\\
\quad \texttt{fun}\ (f : \alpha_2 \to \alpha_3) \mapsto \texttt{return}\ (\\
\quad\quad \texttt{fun}\ (x : \alpha_4) \mapsto (\\
\quad\quad\quad \texttt{do}\ b \leftarrow p\,(x \rhd \omega_1\,); \\
\quad\quad\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad\quad\quad (f\,(x \rhd \omega_2\,)) \rhd \omega_3 \\
\quad\quad\quad \texttt{else} \\
\quad\quad\quad\quad \texttt{return}\ (x \rhd \omega_4\,) \\
)))
\end{array}
$$

2019-2020

EEFF

JFP **30**, e13, 27 pages, 2020.
doi:10.1017/S0956796819000212

Local algebraic effect theories

ŽIGA LUKŠIČ AND MATIJA PRETNAR†

Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
(*e-mails:* ziga.luksic@fmf.uni-lj.si, matija.pretnar@fmf.uni-lj.si)

Abstract

Algebraic effects are computational effects that can be described with a set of basic operations and equations between them. As many interesting effect handlers do not respect these equations, most approaches assume a trivial theory, sacrificing both reasoning power and safety. We present an alternative approach where the type system tracks equations that are observed in subparts of the program, yielding a sound and flexible logic, and paving a way for practical optimisations and reasoning tools.

1 Introduction

Algebraic effects are computational effects that can be described by a *signature* of primitive operations and a collection of equations between them (Plotkin & Power, 2001, 2003), while algebraic effect *handlers* are a generalisation of exception handlers to arbitrary algebraic effects (Plotkin & Pretnar, 2009, 2013). Even though the early work considered only handlers that respect equations of the effect theory, a considerable amount of useful handlers did not, and the restriction was dropped in most—though not all (Ahman, 2017, 2018)—of the later work on handlers (Kammar *et al.*, 2013; Bauer & Pretnar, 2015; Leijen, 2017; Biernacki *et al.*, 2018), resulting in a weaker reasoning logic and imprecise specifications.

Our aim is to rectify this by reintroducing effect theories into the type system, tracking equations observed in parts of a program. On one hand, the induced logic allows us to rewrite computations into equivalent ones with respect to the effect theory, while on the other hand, the type system enforces that handlers preserve equivalences, further specifying their behaviour. After an informal overview in Section 2, we proceed as follows:

- The syntax of the working language, its operational semantics, and the typing rules are given in Section 3.
- Determining if a handler respects an effect theory is in general undecidable (Plotkin & Pretnar, 2013), so there is no canonical way of defining such a judgement. Therefore, the typing rules are given parametric to a reasoning logic, and in Section 4, we present some of the most interesting choices.

† This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

JFP **30**, e13, 27 pages, 2020. © The Author(s) 2020. Published by Cambridge University Press
doi:10.1017/S0956796819000212

Local algebraic effect theories

ŽIGA LUKŠIČ AND MATIJA PRETNAR†
Faculty of Mathematics and Physics, University of Ljubljana, Slovenia
(ziga.luksic@fmf.uni-lj.si, matija.pretnar@fmf.uni-lj.si)

```
theory eqn_assoc for {Choice} is
  { . ; z1 : unit -> *, z2 : unit -> *, z3 : unit -> * |-
    Choice(); b.
      if b then z1 ()
      else Choice(); b'. if b' then z2 () else z3 ()))
  ~
    Choice(); b.
      if b then Choice(); b'. if b' then z1 () else z2 ())
      else z3 ())  }

let to_list : int!eqn_assoc => int list = handler
  | effect Choice _ k -> k true @ k false
  | val x -> [x]
```

Section 4, we present some

† This material is based upon work supported by the Air Force Office of Scientific Research under award number FA9550-17-1-0326.

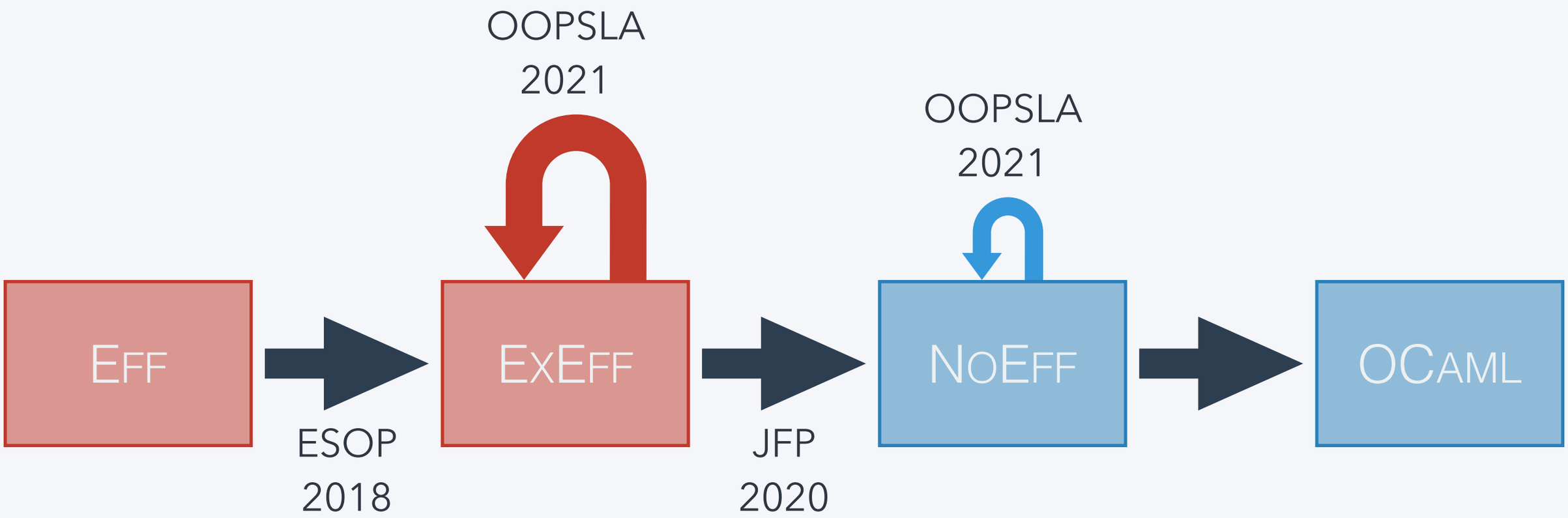
2020

MAJOR CLEANUP

```
let apply (exp1, exp2) =
  match exp1.ty.term with
  | Type.Arrow (ty1, drty2) ->
      assert (Type.equal_ty exp2.ty ty1);
      { term = Apply (exp1, exp2); ty = drty2 }
  | _ -> assert false
```

2021

OPTIMIZATIONS

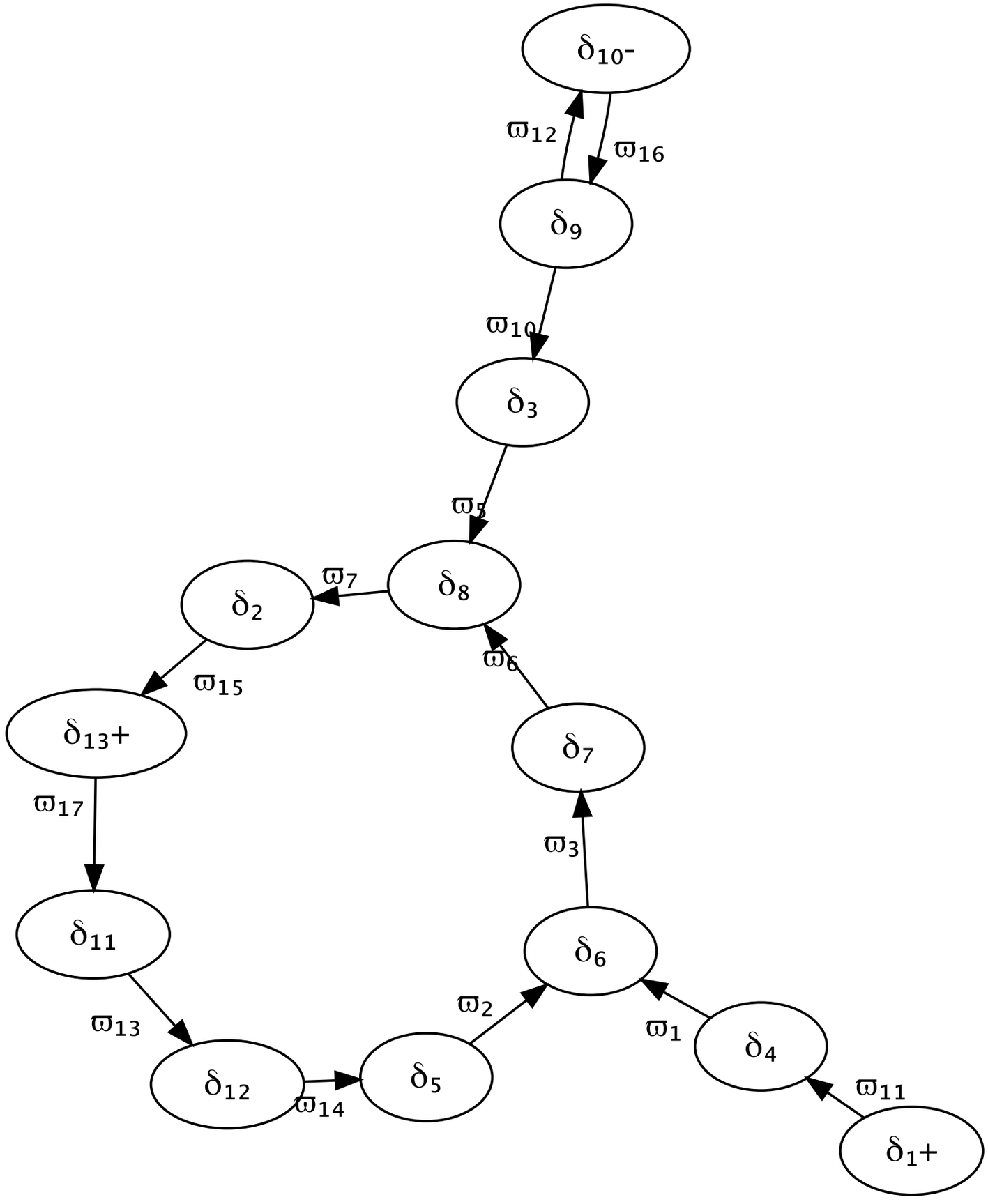

OOPSLA
2021
OOPSLA
2021
Eff
ExEff
NoEff
OCaml
ESOP
2018
JFP
2020

2022-2023

SIMPLIFYING COERCIONS

$$
\begin{array}{l}
\texttt{fun}\ (p : \alpha_1 \to \texttt{bool}) \mapsto \texttt{return}\ (\\
\quad \texttt{fun}\ (f : \alpha_2 \to \alpha_3) \mapsto \texttt{return}\ (\\
\quad\quad \texttt{fun}\ (x : \alpha_4) \mapsto (\\
\quad\quad\quad \texttt{do}\ b \leftarrow p\,(x \rhd \omega_1\,); \\
\quad\quad\quad \texttt{if}\ b\ \texttt{then} \\
\quad\quad\quad\quad (f\,(x \rhd \omega_2\,)) \rhd \omega_3 \\
\quad\quad\quad \texttt{else} \\
\quad\quad\quad\quad \texttt{return}\ (x \rhd \omega_4\,) \\
)))
\end{array}
$$

```
let apply_if w1 w2 w3 w4 p f x =
  p (x |> w1) >>= fun b ->
  if b then
    (f (x |> w2)) |> coer_comp w3
  else
    return (x |> w4)
```


δ10-
ϖ12
ϖ16
δ9
ϖ10
δ3
ϖ5
δ8
ϖ7
δ2
ϖ6
ϖ15
δ13+
δ7
ϖ17
ϖ3
δ11
δ6
ϖ2
ϖ13
ϖ1
δ12
δ5
δ4
ϖ14
ϖ11
δ1+

OPTIMISING SUBTYPING COERCIONS IN A POLYMORPHIC CALCULUS WITH EFFECTS

FILIP KOPRIVEC [a,b] AND MATIJA PRETNAR [a,b]

[a] University of Ljubljana, Faculty of Mathematics and Physics, Jadranska 19, SI-1000 Ljubljana, Slovenia

[b] Institute of Mathematics, Physics and Mechanics, Jadranska 19, SI-1000 Ljubljana, Slovenia
e-mail address: filip.koprivec@imfm.si
e-mail address: matija.pretnar@fmf.uni-lj.si

Abstract.
Write abstract.

write than

write key-words

Before sumbitting go throug LMCS ch list

check if th references are correct

Introduction

Recent years have seen an increase in the number of programming languages that support algebraic effect handlers [PP03, PP13]. With a widespread usage, the need for performance is becoming ever more important. And there are two main ways for achieving it: . an efficient runtime [DWS+15, SDW+21], or an optimising compiler [SBO20, XL21, KKPS21], which we focus on in this paper.

Our recent work [KKPS21] has shown how an optimising compiler can take code written using the full flexibility of handlers, infer precise information about which parts of it use effects and which are pure, and produce code that matches conventional handcrafted one. However, the approach tracks effect information through explicit subtyping coercions [], and for polymorphic functions, these need to be passed around as additional parameters. Since subtyping coercions are inferred automatically, their number grows with the size of the program, which drastically reduces the performance. To avoid this, all polymorphic functions need to be annotated with particular types, or monomorphised by the compiler, neither of which is a satisfactory solution.

In this paper, we propose an algorithm that drastically, yet soundly, reduces (and often completely eliminates) redundant coercion parameters, leading to a performance comparable to monomorphic code. We start with an overview of the approach (Section 1) and continue with a specification of our working language (Section 2). Afterwards, we turn to our contributions, which are:

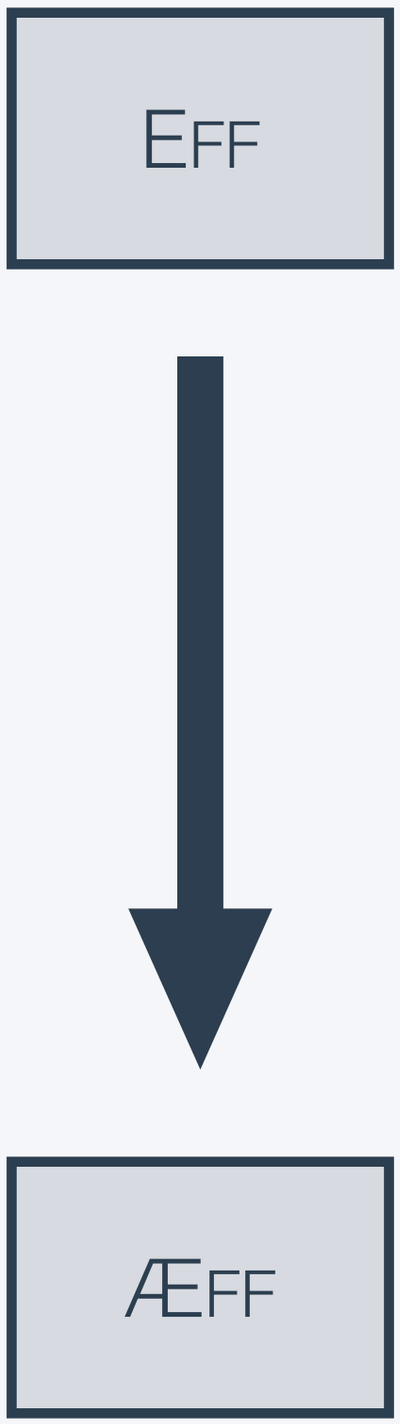
- Identifying requirements for a simplification algorithm phase to be correct with respect to typing (Section **??**).

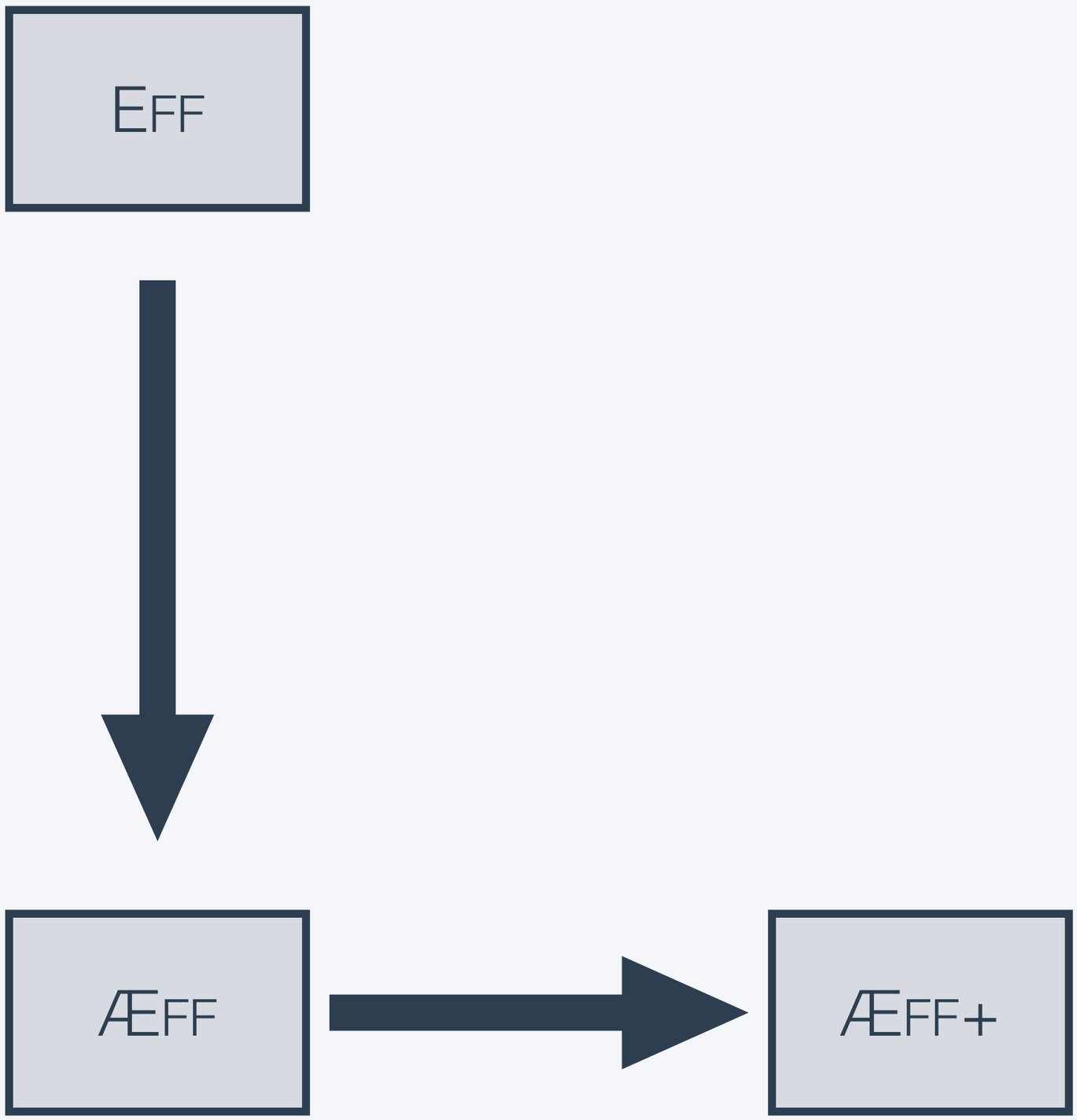
Key words and phrases: Computational effects, Optimizing compilation, Polymorphic compilation, Denotational semantics.

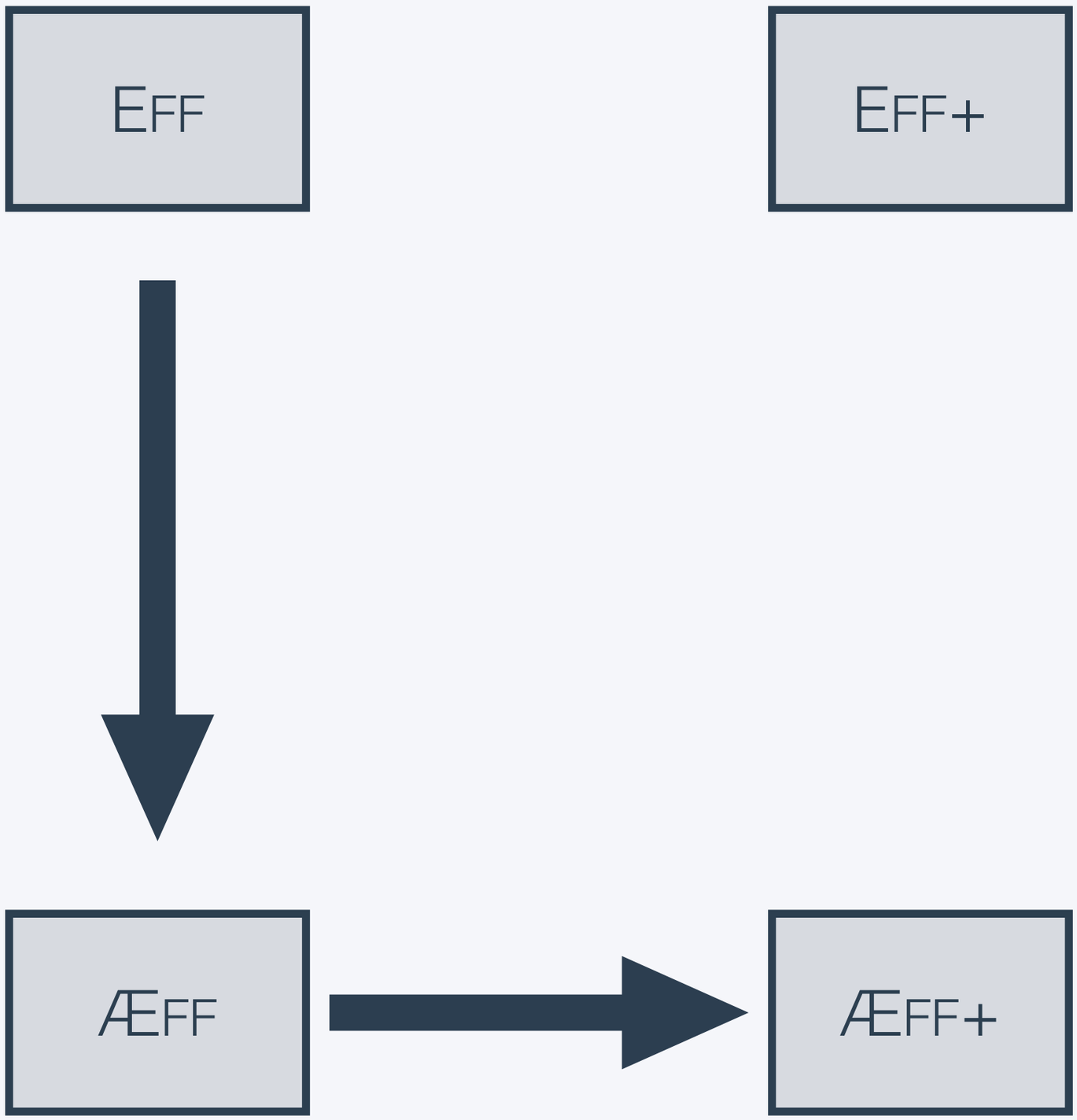
This material is based upon work supported by the Air Force Office of Scientific Research under awards number FA9550-17-1-0326 and FA9550-21-1-0024.

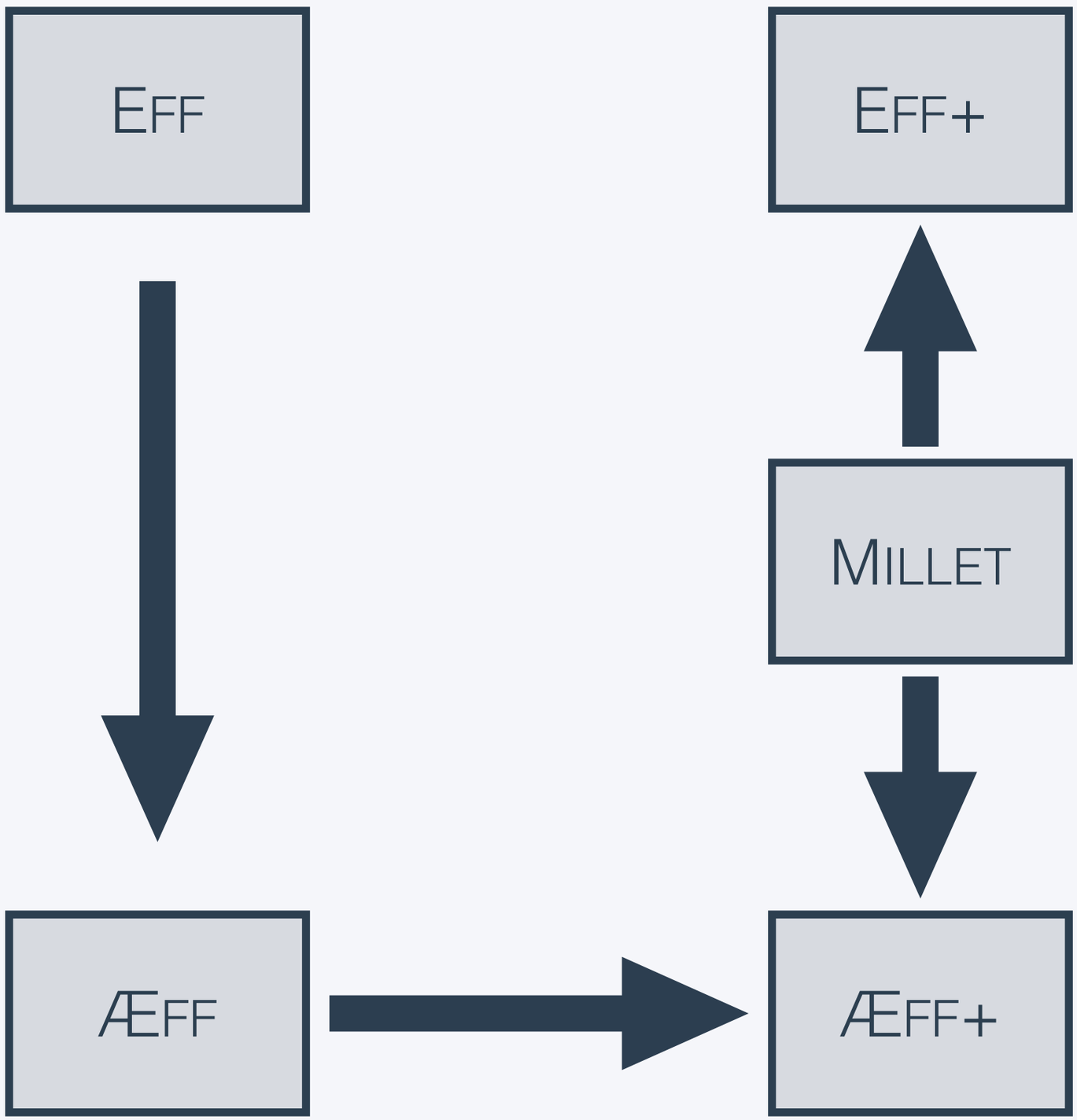
2024-

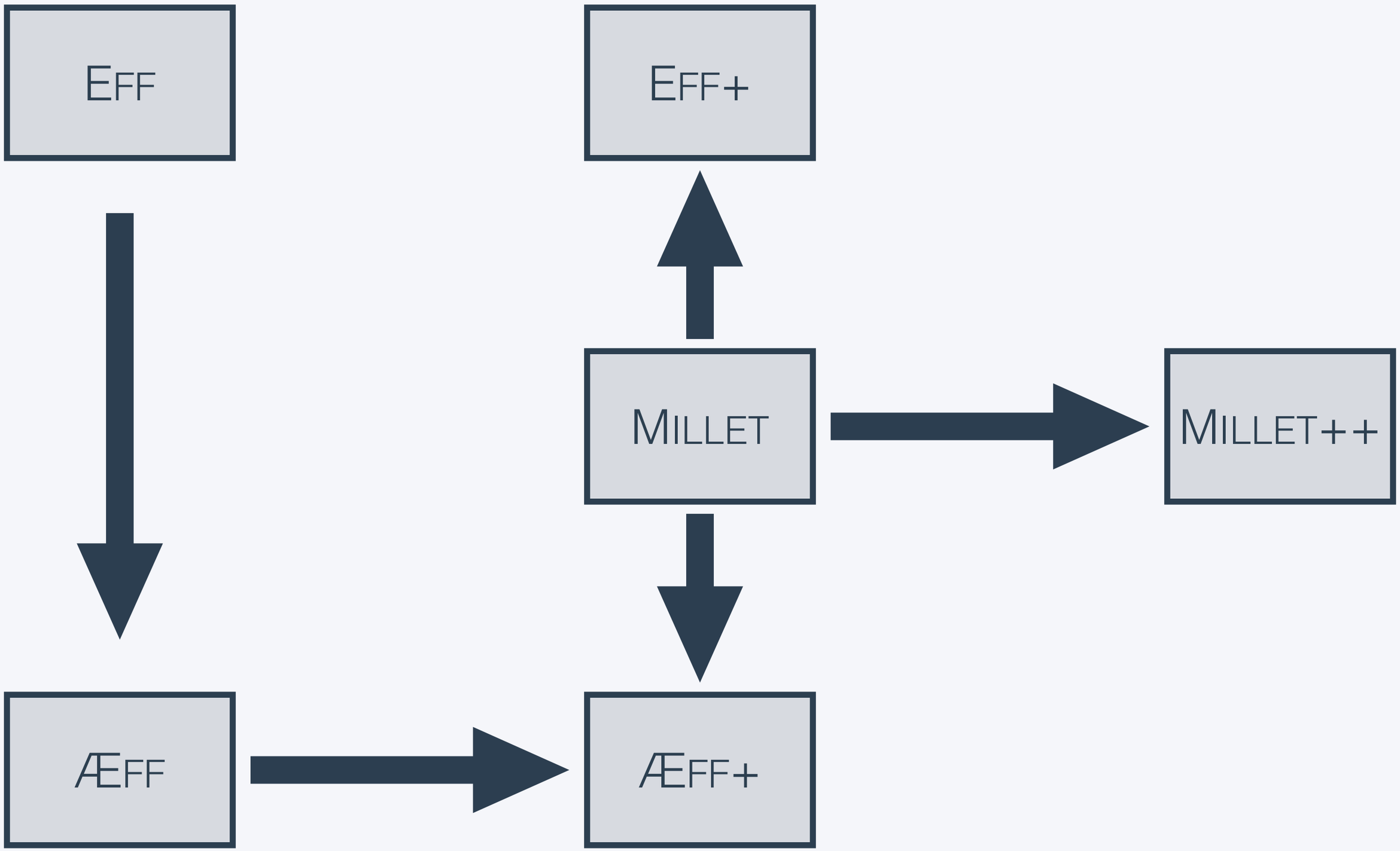
Eff

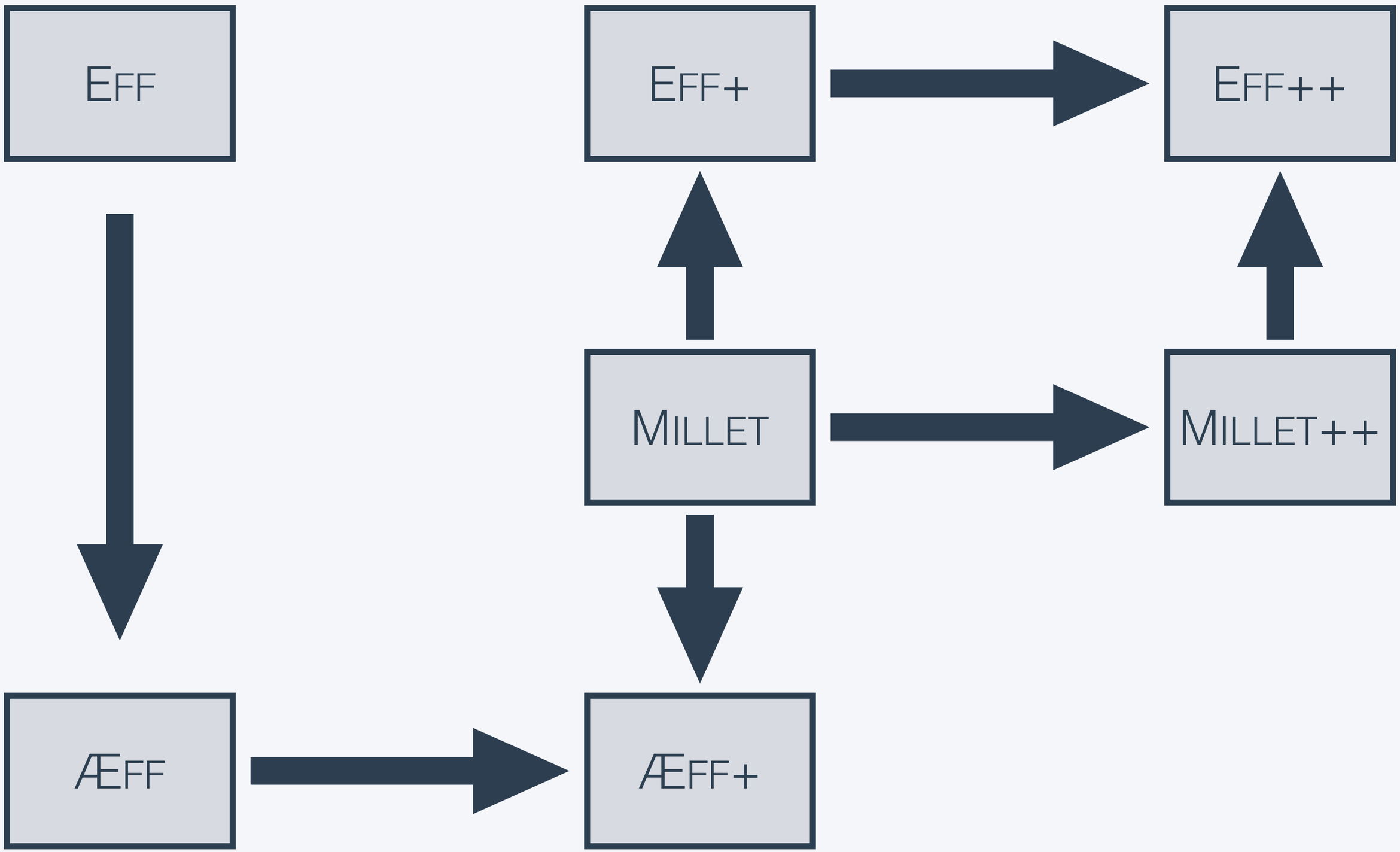

Eff
Æff

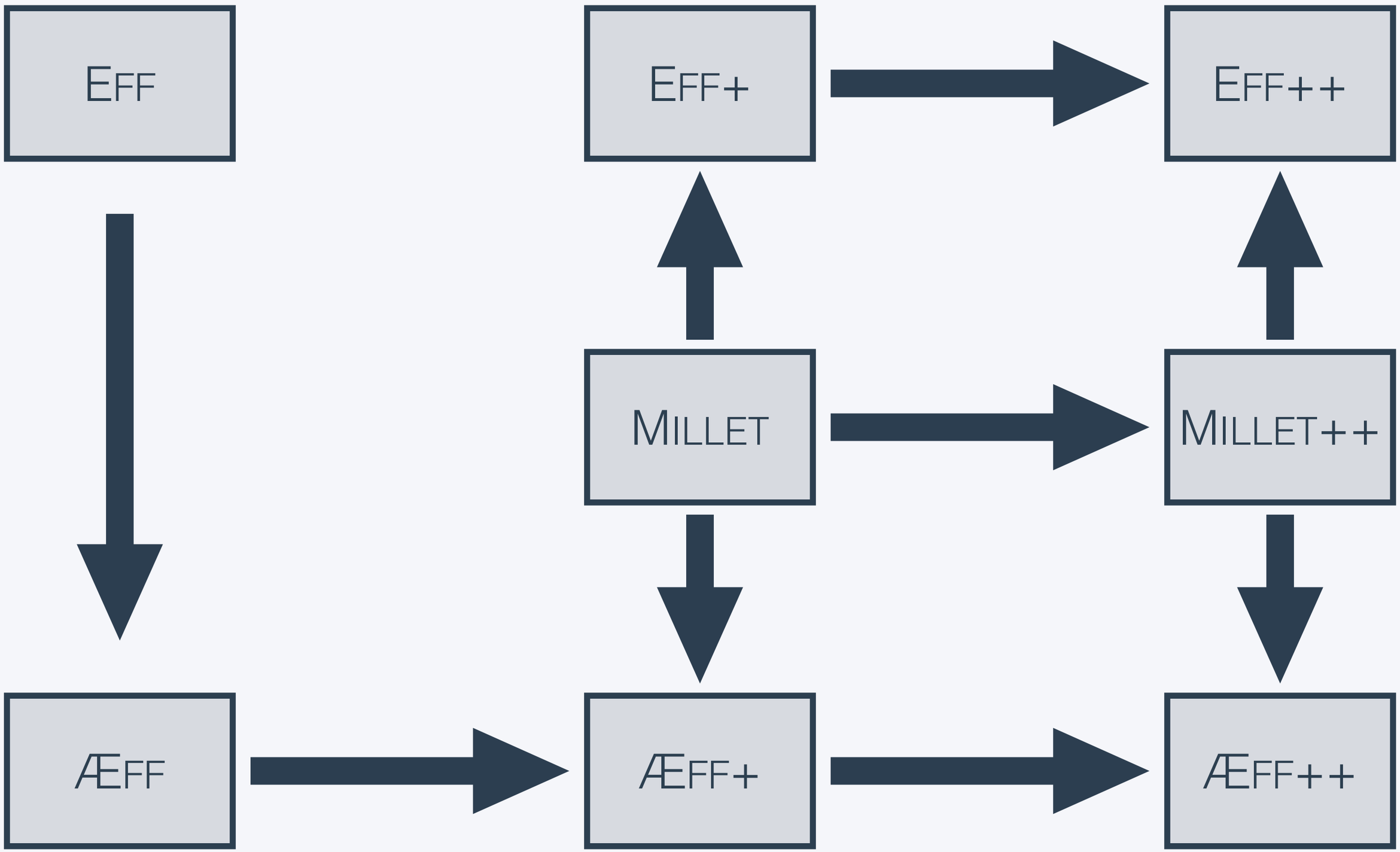

Eff
Æff
Æff+


Eff
Eff+
Æff
Æff+


Eff
Eff+
Millet
Æff
Æff+


Eff
Eff+
Millet
Millet++
Æff
Æff+


Eff
Eff+
Eff++
Millet
Millet++
Æff
Æff+


Eff
Eff+
Eff++
Millet
Millet++
Æff
Æff+
Æff++