Polymorphic Types and Effects with Boolean Unification

MAGNUS MADSEN, Aarhus University, Denmark
JACO VAN DE POL, Aarhus University, Denmark

We present a simple, practical, and expressive type and effect system based on Boolean constraints. The effect system extends the Hindley-Milner type system, supports parametric polymorphism, and preserves principal types modulo Boolean equivalence. We show how to support type inference by extending Algorithm W with Boolean unification based on the successive variable elimination algorithm.

We implement the type and effect system in the Flix programming language. We perform an in-depth evaluation on the impact of Boolean unification on type inference time and end-to-end compilation time. While the computational complexity of Boolean unification is NP-hard, the experimental results demonstrate that it works well in practice. We find that the impact on type inference time is on average a 1.4x slowdown and the overall impact on end-to-end compilation time is a 1.1x slowdown.

CCS Concepts: • **Theory of computation → Type theory**; • **Software and its engineering → General programming languages**.

Additional Key Words and Phrases: polymorphic types and effects, Boolean unification, type inference

ACM Reference Format:
Magnus Madsen and Jaco van de Pol. 2020. Polymorphic Types and Effects with Boolean Unification. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 154 (November 2020), 29 pages. https://doi.org/10.1145/3428222

1 INTRODUCTION

A type and effect system characterizes the values and the computational effects of an expression. While type systems are common in mainstream programming languages, effect systems have seen less use. In this paper, we present a simple, practical, and expressive effect system that tracks purity, but our framework can also be instantiated with other types of computational effects.

A major challenge for type and effect systems is effect polymorphism [Leijen 2014; Lucassen and Gifford 1988; Rytz et al. 2012]. The problem is the following: for higher-order functions the effect of a function depends on the effects of its arguments. For example, if `map` is passed a pure function f then the expression `List.map(f, 1 :: Nil)` is pure. On the other hand, if `map` is passed an impure function g then the expression `List.map(g, 1 :: Nil)` is impure. The effect of `map` depends on the effect of its first argument: it is effect polymorphic.

In this paper, we present a simple, practical, and expressive type and effect system based on Boolean constraints. The effect system is polymorphic and captures whether an expression is pure, impure, or effect polymorphic. We formulate the effect system as an extension of Hindley-Milner [Damas 1984; Hindley 1969; Milner 1978] that preserves principal types. That is, every well-typed expression has a most general type modulo Boolean equivalence. We show how to

Authors' addresses: Magnus Madsen, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark, magnusm@cs.au.dk; Jaco van de Pol, Department of Computer Science, Aarhus University, Åbogade 34, Aarhus, 8210, Denmark, jaco@cs.au.dk.

2475-1421/2020/11-ART154
https://doi.org/10.1145/3428222

support type and effect inference with an extension of Algorithm W that uses Boolean unification based on the successive variable elimination algorithm [Boole 1847; Martin and Nipkow 1989].

The type and effect system is fine-grained and expressive. By fine-grained, we mean that it precisely describes the effect of every expression. By expressive, we mean that it can capture constraints that are not captured by existing effect systems. For example, we can express that a higher-order function takes two function arguments of which at most one is impure. We give some examples that demonstrate that such types are useful in practice.

The standard type inference problem is to compute the most general type of an expression. For a Hindley-Milner style type system, the problem can be solved by unification based on Algorithm W. In this paper, we work in a richer setting: we allow the programmer to provide more specific type annotations on let-bindings to control the degree of polymorphism. That is, we are interested in type inference *subject to additional type and effect constraints*. An important difference between our effect system and many other effect systems is that we do *not* permit sub-effecting: a pure function *cannot* be used where an impure function is required. This allows programmers to specify that a function requires pure as well as impure arguments.

We implement the effect system as an extension of the Flix programming language[1]. Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, parametric polymorphism, currying, higher-order functions, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [Madsen and Lhoták 2018; Madsen et al. 2016].

Given that the computational complexity of Boolean unification is NP-hard [Baader 1998; Martin and Nipkow 1989], a major concern is whether a type and effect system based on such an approach is practical for a real-world programming language. To investigate this question, we conduct an in-depth evaluation of the cost of adding the effect system to the type inference in Flix.

The most important findings from the experimental results are: (i) pure, impure, and effect polymorphic functions are common throughout the Flix standard library, (ii) the computed types with effects are only moderately larger than those without effects (in particular there is no exponential blow-up of Boolean formulas), (iii) the use of Boolean simplifications during successive variable elimination is essential, (iv) the impact on type inference time is on average a 1.4x slowdown, and (v) the overall impact on end-to-end compilation time is a 1.1x slowdown.

In summary, the contributions of the paper are:

- (**Type and Effect System**) We present a polymorphic type and effect system with Boolean constraints. The type and effect system extends Hindley-Milner and preserves principal types. We show how to support type inference by extending Algorithm W with Boolean unification based on the successive variable elimination algorithm.
- (**Implementation**) We implement the type and effect system in the Flix programming language. We refactor the Flix standard library with appropriate effect annotations. These annotations prevent users from unintended usage of the library functions.
- (**Performance Evaluation**) We perform an in-depth evaluation on the impact of Boolean unification on type inference time and end-to-end compilation time. Our most important findings are: (i) the impact on type inference time is on average a 1.4x slowdown, and (ii) the overall impact on end-to-end compilation time is a 1.1x slowdown.

In this work, we realize our goal of designing a simple and expressive effect system based on Boolean constraints that can be implemented in a realistic programming language. Our experiments supports that embedding an NP-hard subroutine as part of a type inference algorithm is practical.

[1] https://flix.dev/

2 MOTIVATION

We motivate our work by showing how the type and effect system can separate pure, impure, and effect polymorphic functions. We give examples of when to enforce purity and impurity. These design choices reflect the changes we have made to the Flix standard library, as discussed in Section 5. Readers who are already familiar with type and effect systems may skip to Section 2.2.

2.1 Motivating Examples

Example I. We can enforce that the predicate f passed to the function Set.exists is pure:

```
def exists(f: a -> Bool, xs: Set[a]): Bool & Pure = ...
```

The signature $f : a \to \text{Bool}$, where a is a type variable, denotes a *pure* function from a to Bool. Passing an impure function to Set.exists is a compile-time type error. We enforce that f is pure because the contract for Set.exists makes no guarantees about how f is called: the implementation of Set.exists may call f on the elements in `xs` in any order and any number of times. This requirement is beneficial because it allows freedom in the implementation of Set, including in the choice of the underlying data structure and in the implementation of its operations. For example, we can implement sets using search trees or with hash tables, and we can perform existential queries in parallel. If f was impure such implementation details would leak and be observable to the client.

Example II. We can enforce that the function f passed to the function List.foreach is impure:

```
def foreach(f: a ~> Unit, xs: List[a]): Unit & Impure = ...
```

The signature $f : a \rightsquigarrow \text{Unit}$ denotes an *impure* function from a to Unit. Passing a pure function to List.foreach is a compile-time type error. Given that f is impure and f is called within List.foreach, it must itself also be impure. This is indicated with the annotation `Unit & Impure` that specifies the return type and effect of the function. If no effect annotation is given, the function is implicitly marked as pure, but for the sake of the exposition, we will keep using explicit annotations. We enforce that f is impure because it is pointless to apply a pure function with Unit return type to every element of a list. While such behavior may be seen as harmless, we want our type and effect system to help avoid programming mistakes [Xie and Engler 2002].

Example III. We can enforce that event listeners are impure:

```
def onMouseDown(f: MouseEvent ~> Unit): Unit & Impure = ...
def onMouseUp(f: MouseEvent ~> Unit): Unit & Impure = ...
```

We enforce that event listeners are impure since they are always executed for their side-effect.

Example IV. We can enforce that assertion and logging facilities are given pure functions[2]:

```
def assert(f: Unit -> Bool): Unit & Pure = ...
def log(f: Unit -> String, l: LogLevel): Unit & Pure = ...
```

We want to support assertions and log statements that can be enabled and disabled at run-time. For efficiency, it is critical that when assertions or logging is disabled, we do not perform any computations that are redundant. We can achieve this by having the assert and log functions take callbacks that are only invoked when required. A critical property of these functions is that they must not influence the execution of the program. Otherwise, we risk situations where enabling or disabling assertions or logging may impact the presence or absence of a buggy execution. We can prevent such situations by requiring that the functions passed to assert and log are pure.

[2]We have marked assert as pure even though it may trigger a process panic. This is because an assertion failure is a side-effect outside the scope of normal execution. We have also marked log as pure, but for a different reason: we consider its side-effect to be *benign*. In a real system, such pragmatism is often required.

Example V. We can enforce that user-defined equality functions are pure[3]:

```
trait Eq[a] {
  def eq(x: a, y: a): Bool & Pure
}
```

We want to enforce that user-defined equality is pure because the programmer should not make any assumptions about how such functions are used. Moreover, most collections (e.g. sets and maps) require that equality does not change over time to maintain internal data structure invariants. Similarly, we can enforce that hashing and comparator functions are pure:

```
trait Hash[a] {
  def hash(x: a): Int32 & Pure
}
```

```
trait Ord[a] {
  def compare(x: a, y: a): Int & Pure
}
```

Example VI. We can also enforce that one-shot comparator functions are pure:

```
def minBy(f: a -> b, l: List[a]): a = ...
def maxBy(f: a -> b, l: List[a]): a = ...
def sortBy(f: a -> Int32, l: List[a]): List[a] = ...
def groupBy(f: a -> k, l: List[a]): Map[k, List[a]] = ...
```

Example VII. We can enforce that expressions that are passed to spawn are impure. For example, we can ensure that the following program is *rejected*:

```
spawn (2 + 2)
```

We want to reject such programs because it is useless to evaluate a pure expression in a separate process implicitly discarding its result. A spawned process must have a side-effect (e.g. communication over a channel).

Example VIII. We can enforce that the function next passed to List.unfoldWithIter is impure:

```
def unfoldWithIter(next: Unit ~> Option[a]): List[a] & Impure
```

The unfoldWithIter function is a variant of the unfoldWith function where each invocation of next changes some mutable state until the unfold completes. For example, unfoldWithIter is frequently used to convert Java-style iterators into lists. We enforce that next is impure since otherwise the iterator cannot advance.

Example IX. We can reject statement expressions that are pure. For example, the program:

```
def main(): Int =
    List.map(x -> x + 1, 1 :: 2 :: Nil);
    123
```

is rejected with the compile-time error:

```
-- Redundancy Error ------------------ foo.flix

>> Useless expression: It has no side-effect(s)
   and its result is discarded.

2 |     List.map(x -> x + 1, 1 :: 2 :: Nil);
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
        useless expression.
```

[3]Flix does not yet have type classes, but they are being implemented.

In the OCaml programming language, the typing rule for the sequence expression $e_1; e_2$ mandates that e_1 must have Unit type. While this prevents any value (other than Unit) from being discarded, it forces the programmer to write `let _ = f()` bindings whenever a non-Unit result of a function call must be discarded. This requirement seems too strict. For example, most mutable set implementations have an add function that adds an element to the set and returns `true` if the element was not already present in the set. Using such a function in OCaml seems unnecessarily cumbersome. In our extension of Flix, we instead require that the expression e_1 is impure.

2.2 Effect Polymorphism

The type and effect system we propose supports *effect polymorphism* [Lucassen and Gifford 1988]. That is, the effect of a higher-order function may depend on the effect of its function argument(s). Effect polymorphism is crucial for reuse of polymorphic code in both pure and impure contexts. We write $a \xrightarrow{e} b$ as the type of a function that produces a result of type b, with a latent effect e, when applied to an argument of type a. In code, we write this as `f: a -> b & e`.

Example X. We can express that the List.map function is *effect polymorphic* in its argument:

```
def map(f: a -> b & e, xs: List[a]): List[b] & e = ...
```

The syntax $f : a \to b \,\&\, e$ denotes a function from a to b with latent effect e. The signature of the List.map function captures that its effect e depends on the effect of its argument f. That is, if List.map is called with a pure function then its evaluation is pure, whereas if it is called with an impure function then its evaluation is impure.

Example XI. We can express that forward function composition » is effect polymorphic:

```
def >>(f: a -> b & e1, g: b -> c & e2): a -> c & (e1 and e2) = x -> g(f(x))
```

The » function takes two effect polymorphic functions $f : a \xrightarrow{e_1} b$ and $g : b \xrightarrow{e_2} c$ with effects e_1 and e_2, and returns a function $a \xrightarrow{e_1 \wedge e_2} c$. The returned function is pure if and only if f and g are both pure, i.e. $e_1 \wedge e_2$. The ability to combine effects is important and is used for several other functions in the Flix standard library.

Example XII. In a pure functional programming language the following identity holds:

```
map(f, map(g, l)) == map(f >> g, l)
```

for two functions f and g, and a list l. In words, mapping g over the list l and then mapping f over the resulting list is equivalent to mapping the forward function composition $f >> g$ over the list l.

A combinator library might want to take advantage of this situation by offering a function like:

```
def mapCompose(f: a -> b, g: b -> c, l: List[a]): List[c]
```

to enable more efficient list processing. Alternatively, a compiler might come equipped with a set of rewrite rules that perform similar optimizations [Jones et al. 2001]. Unfortunately, in an impure programming language like Flix, the above identity is not true in general. For example, the program:

```
let f = x -> {Console.printLine(x); x};
let g = y -> {Console.printLine(y); y};
List.map(f, List.map(g, 1 :: 2 :: Nil))
```

prints the sequence 1, 2, 1, 2. But if f and g are composed the program prints 1, 1, 2, 2. Consequently, we cannot perform the desired rewrite. If we restrict f and g to pure functions then we are back to the Haskell-situation. However, this requirement is too strict: it is sufficient if *at most* one of the f and g functions is impure. We can express this in our type and effect system!

```
def mapCompose(f: a -> b & e1, g: b -> c & (not e1 or e2), l: List[a]):
               List[c] & (e1 and e2) = ...
```

Let us break this down. For clarity, let us write down the type of mapCompose:

$$\forall a, b, c. \forall e_1, e_2. (a \xrightarrow{e_1} b) \to (b \xrightarrow{\neg e_1 \vee e_2} c) \to \mathsf{List}[a] \to \mathsf{List}[c] \,\&\, e_1 \wedge e_2$$

Let us now consider two cases (where f and g refer to the first and second argument):

- If f is pure ($e_1 = \mathsf{T}$) then $\neg e_1 \vee e_2 = \neg\mathsf{T} \vee e_2 = e_2$. Thus g *may* be pure or impure.
- If f is impure ($e_1 = \mathsf{F}$) then $\neg e_1 \vee e_2 = \neg\mathsf{F} \vee e_2 = \mathsf{T}$. Thus g *must* be pure.

In other words, as desired, at most one of f and g can be impure. To further illustrate, let us see what happens if we try to pass *two* impure functions to mapCompose. In that case, we have: $f : a \xrightarrow{\mathsf{F}} b$ and $g : b \xrightarrow{\mathsf{F}} c$. This implies that the following Boolean equivalences must hold:

$$e_1 \equiv \mathsf{F} \quad \text{and} \quad \neg e_1 \vee e_2 \equiv \mathsf{F}$$

but these constraints are inconsistent because $\mathsf{T} \not\equiv \mathsf{F}$.

In addition to the presented example, we have found such polymorphic effects to be useful for expressing lints and rewrite rules in the style of Jones et al. [2001].

2.3 Type Equivalence and Most General Types Modulo Boolean Equivalence

In the previous example, we had the type:

$$\forall a, b, c. \forall e_1, e_2. (a \xrightarrow{e_1} b) \to (b \xrightarrow{\neg e_1 \vee e_2} c) \to \mathsf{List}[a] \to \mathsf{List}[c] \,\&\, e_1 \wedge e_2$$

but we might as well have written the type:

$$\forall a, b, c. \forall e_1, e_2. (a \xrightarrow{\neg e_1 \vee e_2} b) \to (b \xrightarrow{e_1} c) \to \mathsf{List}[a] \to \mathsf{List}[c] \,\&\, e_1 \wedge e_2$$

where the effects on the two higher-order functions are swapped. In our type and effect system both types are *equally general*. The main theoretical result of our paper is to extend the Hindley-Milner type system with effects modulo Boolean equivalence, and then to show that type inference is still practical. Specifically, we extend Algorithm W with Boolean unification based on the successive variable elimination algorithm. The result rests on the fact that Boolean unification is decidable and admits a most general unifier [Boole 1847; Martin and Nipkow 1989].

2.4 Sub-Effecting vs. Enforcing Purity and Impurity

By design, the type and effect system *does not* allow sub-effecting: pure function types are not subtypes of impure function types. If a higher-order function expects an impure function argument then it is illegal to pass a pure function. While such programs cannot "go wrong" in the traditional sense, we want to enable programmers to express their intent and have it enforced by the type and effect system. In our system, a pure function cannot be passed where an impure is expected, but there is no *guarantee* that an impure function will have an effect at run-time. For example, the following expression is considered impure:

```
1  if (true) 123 else { Console.printLine("Hello World"); 123 }
```

despite the fact that it will never print anything at run-time. In other words, *purity* guarantees the absence of side-effects, whereas *impurity* merely indicates the possibility of side-effects.

3 CALCULUS

We now present λ_{eff} a minimal lambda calculus with a polymorphic type and effect system. We then present an extension of Hindley-Milner style type inference that enables type and effect inference for the calculus. We begin with an introduction to Boolean unification.

3.1 Preliminaries: Boolean Unification

Boolean Formulas. Let $\mathcal{B} = \{\mathsf{T}, \mathsf{F}\}$ denote the Boolean values true and false, and let *BoolVar* denote a countable set of Boolean (proposition) variables ($\beta, \ldots$). The set *Formula* of Boolean formulas ($\varphi, \psi, \ldots$) is then defined inductively by:

$$\varphi, \psi \in \mathit{Formula} = \mathsf{T} \mid \mathsf{F} \mid \beta \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$$

A Boolean formula of n variables denotes a Boolean function $\mathcal{B}^n \to \mathcal{B}$, according to the standard semantics. We write $\varphi \equiv_{\mathcal{B}} \psi$ if the Boolean formulas φ and ψ denote the same Boolean function (i.e., they are logically equivalent). We will write $\varphi \Rightarrow \psi$ if φ logically implies ψ.

Substitutions. A *substitution* S is a partial mapping *BoolVar* $\to$ *Formula*. We write $\{x_i \mapsto \psi_i\}_{i \in I}$ for the substitution S, such that $S(x_i) = \psi_i$. We extend substitutions to a total function on formulae, and write $S(\varphi)$ for the result of applying S to φ. We write $\varphi \leq \psi$ (formula φ is more general than ψ, if there exists a substitution S, such that $S(\varphi) = \psi$. The composition $S_2 \circ S_1$ of substitutions S_1 and S_2 is defined such that $(S_2 \circ S_1)(\varphi) = S_2(S_1(\varphi))$. We will often drop $\circ$ and () and simply write $S_2 S_1 \varphi$. We write $S_1 \leq_{\mathcal{B}} S_2$ (substitution S_1 is more general than S_2) iff there exists a substition S_3 such that $S_3 S_1 \equiv_{\mathcal{B}} S_2$, i.e., for all $x \in \beta$, $S_3 S_1(x) \equiv_{\mathcal{B}} S_2(x)$. Note that composition of substitutions is associative and monotone for $\leq_{\mathcal{B}}$ in both arguments, and note that $\leq_{\mathcal{B}}$ is reflexive and transitive. We write $\varphi_1 \sqsubseteq_{\mathcal{B}} \varphi_2$ if there exists S such that $S\varphi_1 \equiv_{\mathcal{B}} \varphi_2$.

Definition 3.1 (Boolean Unification Problem). Given two Boolean formulas φ and ψ the *Boolean Unification Problem* $\varphi \stackrel{?}{=} \psi$ is to compute a unifier (solution), i.e. a substitution S such that $S(\varphi) \equiv_{\mathcal{B}} S(\psi)$ or to report that no such substitution exists. Similarly, a solution S for a set of Boolean Unification Problems $\{\varphi_i \stackrel{?}{=} \psi_i\}_{i \in I}$ is a single substitution S with $S(\varphi_i) \equiv_{\mathcal{B}} S(\psi_i)$ (for all $i \in I$).

A solution S is a most general unifier (mgu), if $S \leq_{\mathcal{B}} S'$ for all other unifiers S'.

Note that the most general unifier is not unique, not even modulo $\equiv_{\mathcal{B}}$ (cf. Table 1. 8a,b). The point is that the set of all solutions can be represented by a single unifier (i.e., the theory is unitary).

Theorem 3.2 ([Martin and Nipkow 1989]). *Boolean Unification is decidable. Moreover, if $\varphi \stackrel{?}{=} \psi$ is solvable then there exists a most general unifier.*

This result goes back to Boole himself [Boole 1847]. Boole's algorithm is known as the *successive variable elimination* algorithm (SVE). We first provide a couple of examples.

Example 3.3. Table 1 provides several instances of Boolean unification problems $\varphi \stackrel{?}{=} \psi$, yielding a most general unifier S when a solution exists. For all cases, it is easy to verify that S is a *unifier*, i.e. $S(\varphi) \equiv_{\mathcal{B}} S(\psi)$. It is less obvious to check that the result is a *most general* unifier.

For example, (8a) and (8b) seem to yield quite different substitutions $S = \{z \mapsto x \wedge y\}$ and $T = \{x \mapsto (x \wedge \neg y) \vee z,\ y \mapsto y \vee z\}$. However, both are equally general, since $S \leq_{\mathcal{B}} T$ and $T \leq_{\mathcal{B}} S$. First, $S \leq_{\mathcal{B}} T$, since $TS = T$: Clearly, $T(S(x)) = T(x)$ and $T(S(y)) = T(y)$. Finally, $T(S(z)) = T(x \wedge y) = ((x \wedge \neg y) \vee z) \wedge (y \vee z) \equiv_{\mathcal{B}} (x \wedge \neg y \wedge y) \vee z \equiv_{\mathcal{B}} z = T(z)$. Similarly, the reader can check that $ST \equiv_{\mathcal{B}} S$, so $T \leq_{\mathcal{B}} S$ holds as well.

3.2 The Successive Variable Elimination Algorithm

We now present the SVE algorithm, originally due to Boole. It is exponential in the number of Boolean variables. One might worry that this is impractical for real-world type inference. The experimental part of this paper demonstrates otherwise. We reformulate Boole's algorithm as presented in [Martin and Nipkow 1989] for Boolean rings, with operator xor for *exclusive or*. The key insight in the algorithm is that Boolean unification can be reduced to a matching problem:

$\varphi \stackrel{?}{=} \psi$ has the same solutions as $(\varphi \,\mathsf{xor}\, \psi) \stackrel{?}{=} \mathsf{F}$.

Table 1. Examples of Boolean Unification

No.	φ	ψ	mgu S of $\varphi \stackrel{?}{=} \psi$
(1)	$x \wedge y$	T	$\{x \mapsto \mathsf{T}, y \mapsto \mathsf{T}\}$
(2)	$x \wedge y$	F	$\{x \mapsto x \wedge \neg y\}$
(3)	$x \vee y$	T	$\{x \mapsto \neg y \vee x\}$
(4)	$x \vee y$	F	$\{x \mapsto \mathsf{F}, y \mapsto \mathsf{F}\}$
(5)	$x \vee y$	$x \wedge y$	$\{x \mapsto y\}$
(6)	T	F	NO UNIFIER
(7)	x	$\neg x$	NO UNIFIER
(8a)	$x \wedge y$	z	$\{z \mapsto x \wedge y\}$
(8b)	$x \wedge y$	z	$\{x \mapsto (x \wedge \neg y) \vee z,\ y \mapsto y \vee z\}$
(9)	$y \vee \neg(x \wedge z)$	$x \wedge \neg y \wedge z$	NO UNIFIER
(10)	$a \wedge b$	$c \wedge d$	$\{b \mapsto (c \wedge d) \vee b, a \mapsto (c \wedge d) \vee (a \wedge \neg b)\}$

Flix

```
1  def sve(f: Formula): Substitution =
2      if (f == F) { return ∅ }
3      else if (Var(f) = ∅) { throw BooleanUnificationException }
4      else
5          pick x from Var(f)
6          t0 = f[x := F]
7          t1 = f[x := T]
8          S = sve((t0 ∧ t1)↓)
9          return S ∪ {x ↦ S(t0) ∨ (x ∧ ¬S(t1))}
10
11 def unify_B(f1, f2: Formula): Substitution =
12     return sve(f1 xor f2)
```

Fig. 1. The Successive Variable Elimination (SVE) Algorithm used by $\text{unify}_{\mathcal{B}}$.

The following theorem shows how we can eliminate a single variable from a matching problem. We use the traditional notation $\varphi[x := \psi]$ for the substitution application $\{x \mapsto \psi\}(\varphi)$.

Theorem 3.4 ([Martin and Nipkow 1989, Equation 2.12]). *$\varphi \stackrel{?}{=} \mathsf{F}$ has an mgu if and only if $\varphi[x := \mathsf{F}] \wedge \varphi[x := \mathsf{T}] \stackrel{?}{=} \mathsf{F}$ has an mgu S. In that case, the mgu of $\varphi \stackrel{?}{=} \mathsf{F}$ is the composition of substitutions, $S \circ \{x \mapsto \varphi[x := \mathsf{F}] \vee (x \wedge \neg\varphi[x := \mathsf{T}])\}$*

Figure 1 shows Boole's successive variable elimination (SVE) algorithm. It recursively applies Theorem 3.4, eliminating all variables ($\text{Var}(f)$) from f one by one. Appendix A illustrates how the SVE algorithm works.

An exponential blowup could arise because in each recursive call, the formula is doubled in the worst case. In practice, this is avoided by simplifying the intermediate formulas, indicated by $\varphi\downarrow$. Simplification should at least reduce closed terms to T or F. The simplifications are not specified here, but are available in the technical report.

Two generalizations of SVE. During type inference, we will need to solve a sequence of Boolean Unification problems. This can be achieved by using the SVE procedure as follows:

Proposition 3.5. *If S_1 is a mgu for $\varphi_1 \stackrel{?}{=} \psi_1$ and S_2 is a mgu for $S_1(\varphi_2) \stackrel{?}{=} S_1(\psi_2)$ then S_2S_1 is a mgu for the set of equations $\{\varphi_1 \stackrel{?}{=} \psi_1, \varphi_2 \stackrel{?}{=} \psi_2\}$.*

Proof. Assume that S_1 is a mgu for $\varphi_1 \stackrel{?}{=} \psi_1$ and S_2 is a mgu for $S_1(\varphi_2) \stackrel{?}{=} S_1(\psi_2)$.

We first show that S_2S_1 is a solution of both equations. S_1 is a solution, so $S_1(\varphi_1) \equiv_{\mathcal{B}} S_1(\psi_1)$, but then also $S_2S_1(\varphi_1) \equiv_{\mathcal{B}} S_2S_1(\psi_1)$, since $\equiv_{\mathcal{B}}$ is closed under substitution. S_2 is a solution, so $S_2S_1(\varphi_2) \equiv_{\mathcal{B}} S_2S_1(\psi_2)$.

Next, we show that it is the most general solution. Let S be another solution of both equations, i.e. $S(\varphi_1) \equiv_{\mathcal{B}} S(\psi_1)$ and $S(\varphi_2) \equiv_{\mathcal{B}} S(\psi_2)$. Since S_1 was most general, $S_1 \leq_{\mathcal{B}} S$, i.e., there exists S', such that $S'S_1 \equiv_{\mathcal{B}} S$. So $S'S_1(\varphi_2) \equiv_{\mathcal{B}} S'S_1(\psi_2)$. Since S_2 was most general, $S_2 \leq_{\mathcal{B}} S'$, i.e., there exists S'', with $S''S_2 \equiv_{\mathcal{B}} S'$. But then $S''(S_2S_1) \equiv_{\mathcal{B}} (S''S_2)S_1 \equiv_{\mathcal{B}} S'S_1 \equiv_{\mathcal{B}} S$, so $S_2S_1 \leq_{\mathcal{B}} S$. □

Finally, to compare two type schemes in Section 3.5 we will need a more general procedure, where only some Boolean variables can be instantiated (flexible), while other Boolean variables should be treated as constants (rigid). In particular, we will need to solve problems of the form $\forall Y.\exists X.\varphi(X, Y) \equiv_{\mathcal{B}} \psi(X, Y)$. This can be viewed as the unification problem $\varphi(X?, Y) \stackrel{?}{=} \psi(X?, Y)$, with flexible variables $X?$ and rigid variables Y.

Fortunately, the SVE procedure already computes an mgu for arbitrary Boolean Algebras, not just $\mathcal{B}$, see [Martin and Nipkow 1989]. To solve the more general problem, we work in the Boolean Algebra, whose elements are formulas over the rigid symbols Y. The variables used in unifications problems over this algebra are from $X?$. The test $f == \mathsf{F}$ should now be replaced by the test $f(Y) \equiv_{\mathcal{B}} \mathsf{F}$. This can be checked with a standard SAT solver, viewing Y as Boolean variables: $f(Y)$ is unsatisfiable if and only if $f(Y) \equiv_{\mathcal{B}} \mathsf{F}$. In our implementation, we reuse the SVE procedure on $\mathcal{B}$ to solve this SAT problem.

Example 3.6. $x \wedge y \stackrel{?}{=} z?$ yields the solution 8a (cf. Table 1), $x? \wedge y? \stackrel{?}{=} z$ yields solution 8b, while $x? \wedge y \stackrel{?}{=} z$ has no unifier, since no x satisfies the equation in case $y = \mathsf{F}$ and $z = \mathsf{T}$.

3.3 The λ_{eff} Calculus

Syntax. The syntax of λ_{eff} includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function application. The sequence expression $e_1; e_2$ evaluates e_1, discards its result, and then evaluates e_2. The calculus has two let-expressions: $\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2$ and $\mathsf{let}\, x : \sigma = e_1\, \mathsf{in}\, e_2$. The former does full generalization of e_1 whereas the latter allows the programmer to provide a *closed* type annotation σ that controls the degree of polymorphism of e_1. This allows the programmer to require pure or impure function arguments for higher-order functions. The if-then-else and print expressions are straightforward; they are included to illustrate the effect behavior of the calculus. Figure 2a shows the syntax of λ_{eff}. Note that we could have defined print as a constant $\mathsf{print} : \mathsf{String} \xrightarrow{\mathsf{F}} \mathsf{Unit}$.

Types. The types of λ_{eff} are separated into mono types (τ) and type schemes (σ). The mono types include type variables α, a set of base types ι (e.g. Unit, Bool), 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. That is, the effect φ happens when the function is applied. The language of effects φ is a Boolean formula (recall that a function with effect T is pure). The type schemes include mono types τ and quantified types $\forall\alpha.\sigma$ and $\forall\beta.\sigma$, where α is a type variable and β is an effect variable. Note that every type scheme σ is of the form $\sigma = \forall\overline{\gamma}.\tau$, where $\overline{\gamma}$ is a sequence of type and effect variables. Figure 2b shows the types of λ_{eff}. Boolean equivalence on types is the smallest relation $\equiv_{\mathcal{B}}$ such that:

- If $\tau = \tau'$ then $\tau \equiv_{\mathcal{B}} \tau'$.
- If $\tau_1 \equiv_{\mathcal{B}} \tau_1'$, $\tau_2 \equiv_{\mathcal{B}} \tau_2'$ and $\varphi \equiv_{\mathcal{B}} \varphi'$, then $\tau_1 \xrightarrow{\varphi} \tau_2 \equiv_{\mathcal{B}} \tau_1' \xrightarrow{\varphi'} \tau_2'$.

For example, we have that: $(\tau \xrightarrow{\mathsf{F}} \tau) \xrightarrow{\mathsf{T}} \tau \equiv_{\mathcal{B}} (\tau \xrightarrow{\beta \wedge \neg\beta} \tau) \xrightarrow{\mathsf{T}} \tau$.

Type Substitutions. A substitution on types is a partial function $S : (\mathit{TypeVar} \to \mathit{Type}) \cup (\mathit{BoolVar} \to \mathit{Formula})$, mapping type variables to types and effect variables to effects. We write $\tau \leq \tau'$ (the type τ is more general then τ') if there is a substitution S such that $S(\tau) = \tau'$. We write $\sigma \sqsubseteq \tau'$ (the type τ' is an instance of type scheme σ) if $\sigma = \forall \overline{\gamma}\, \tau$ and there exists a substitution S such that $\mathrm{dom}(S) = \overline{\gamma}$ and $S(\tau) = \tau'$. Finally, we write $\sigma \sqsubseteq_{\mathcal{B}} \sigma'$ (type scheme σ is more liberal than type scheme σ') if every instance of σ' is an instance of σ modulo Boolean equivalence, i.e. if for all τ' with $\sigma' \sqsubseteq \tau'$, there exists a τ with $\sigma \sqsubseteq \tau$ and $\tau \equiv_{\mathcal{B}} \tau'$. We refer to Section 2.3 for an example.

$$
\begin{array}{rcll}
c \in \mathit{Const} & = & \mathsf{T} \mid \mathsf{F} \mid \mathsf{skip} \mid \cdots & \\
v \in \mathit{Val} & = & c \mid \lambda x.\, e & \\
e \in \mathit{Exp} & = & x \mid v \mid e\, e \mid e; e & \\
& \mid & \mathsf{let}\ x = e\ \mathsf{in}\ e & \\
& \mid & \mathsf{let}\ x : \sigma = e\ \mathsf{in}\ e & (\sigma \text{ closed}) \\
& \mid & \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e & \\
& \mid & \mathsf{print}\ e & \\
x, y \in \mathit{Var} & = & \text{a set of variable symbols} &
\end{array}
$$

(a) Expressions of λ_{eff}.

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

(b) Types of λ_{eff}.

Fig. 2. Syntax and Types of λ_{eff}.

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

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_b x : \tau \,\&\, \mathsf{T}}\ (\text{T-Var}) \qquad \frac{\Gamma \vdash_b e_1 : \tau_1 \,\&\, \mathsf{F} \quad \Gamma \vdash_b e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash_b e_1; e_2 : \tau_2 \,\&\, \mathsf{F}}\ (\text{T-Seq})$$

$$\frac{\mathrm{typeOf}(c) = \sigma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash_b c : \tau \,\&\, \mathsf{T}}\ (\text{T-Cst}) \qquad \frac{\Gamma \vdash_b e_1 : \tau_1 \,\&\, \varphi_1 \quad \Gamma, x : \mathrm{gen}(\Gamma, \tau_1) \vdash_b e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash_b \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \,\&\, \varphi_1 \wedge \varphi_2}\ (\text{T-Let})$$

$$\frac{\Gamma \vdash_b e : \mathsf{String} \,\&\, \varphi}{\Gamma \vdash_b \mathsf{print}\ e : \mathsf{Unit} \,\&\, \mathsf{F}}\ (\text{T-Prt}) \qquad \frac{\Gamma \vdash_b e_1 : \tau_1 \,\&\, \varphi_1 \quad \mathrm{gen}(\Gamma, \tau_1) \sqsubseteq_{\mathcal{B}} \sigma \quad \Gamma, x : \sigma \vdash_b e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash_b \mathsf{let}\ x : \sigma = e_1\ \mathsf{in}\ e_2 : \tau_2 \,\&\, \varphi_1 \wedge \varphi_2}\ (\text{T-Let-2})$$

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

$$\mathrm{gen}(\Gamma, \tau) = \forall \alpha_1, \cdots, \forall \alpha_n . \forall \beta_1, \cdots, \forall \beta_n . \tau$$

where $\{\alpha_1, \cdots, \alpha_n, \beta_1, \cdots, \beta_n\} = \mathrm{ftv}(\tau) \setminus \mathrm{ftv}(\Gamma)$

We assume that we have at least the constants $\{\mathsf{T} : \mathsf{Bool}, \mathsf{F} : \mathsf{Bool}, \mathsf{skip} : \mathsf{Unit}\}$.

Fig. 3. Type Rules for λ_{eff} with judgements of the form $\Gamma \vdash_b e : \tau \,\&\, \varphi$.

Type Rules. Figure 3 shows the type rules of λ_{eff}. In order to set up the type system, we define a context Γ as a *partial function* of bindings $x : \sigma$. We also define $\mathrm{ftv}(\sigma)$ to be the type variables that occur free in σ, and $\mathrm{ftv}(\Gamma)$ as the union of all free type variables in its range.

A type judgement is of the form $\Gamma \vdash_b e : \tau \,\&\, \varphi$ which states that under type environment Γ, the expression e has type τ and effect φ. We write the turnstyle with subscript b because later we will present another formulation suited for type inference (with subscript w). Except for (T-Eq), the type rules are syntax directed. The soundness criteria of the type and effect system is that a pure expression cannot produce an effect during evaluation, i.e. if $\Gamma \vdash_b e : \tau \,\&\, \mathsf{T}$ then evaluation of e cannot produce an effect. We will make this notion precise in Section 3.4.

The type rules are mostly straightforward and we only discuss the most interesting ones. The (T-Eq) rule states that if an expression e has type τ_1 and effect φ_1 and if τ_1 is equivalent to τ_2 and φ_1 is equivalent to φ_2, then we may conclude that e has type τ_2 and effect φ_2. This rule embodies the notion of types modulo Boolean equivalence. The (T-Abs) and (T-App) rules type lambda abstractions and applications. An abstraction takes the effect of an expression e and moves it onto the arrow type whereas an application releases the latent effect on an arrow type. The (T-Let-1) and (T-Let-2) rules perform let-generalization. The (T-Let-1) rule is standard. The (T-Let-2) rule checks that the type annotation provided by the programmer is sound and uses that type when type checking the body of the let. This rule enables the programmer to control the degree of effect polymorphism. The (T-Seq) reflects the design decision that we wish to rule out expressions with useless sub-expressions, like $17; 21$. The (T-Seq) requires that the first sub-expression is impure. However, this rule doesn't exclude all pathological cases, and it breaks type preservation too: The term $(\mathsf{print}\ \text{"hello"}; 17); 21$ contains a useless sub-expression, but it is well-typed. However, its reduct $17; 21$ is not well-typed. Therefore, we also consider the more liberal rule (T-Seq'). If the system with (T-Seq') is sound, the system with (T-Seq) is also sound, since the latter is more strict.

$$\frac{\Gamma \vdash_b e_1 : \tau_1 \,\&\, \varphi_1 \quad \Gamma \vdash_b e_2 : \tau_2 \,\&\, \varphi_2}{\Gamma \vdash_b e_1; e_2 : \tau_2 \,\&\, \varphi_1 \wedge \varphi_2} \qquad \text{(T-Seq')}$$

3.4 Soundness of the Type System

We show that the type system is sound, i.e. a pure expression $e : \tau \,\&\, \mathsf{T}$ cannot produce an effect during evaluation. The reverse is not true: some expressions evaluate without any effect, but are still typed impure, for instance "$\mathsf{if}\ \mathsf{F}\ \mathsf{then}\ \mathsf{print}\ s; 17\ \mathsf{else}\ 29 : \mathsf{Nat} \,\&\, \mathsf{F}$". (So this could be used as an impure-conversion by the programmer). But we will show that if-then-else is the only reason for incompleteness. To state the result formally, we must first define the operational semantics.

Operational Semantics. We assume standard, left-to-right, eager call-by-value evaluation of λ_{eff}, except for if-then-else, which has short-cutting semantics. See Figure 4 for the reduction rules and the evaluation contexts. We annotate each step with a Boolean: T if the step is pure, and F if the step has a side effect.

Lemma 3.7. *Every closed typeable expression $\vdash_b e : \tau \,\&\, \varphi$ reduces to a value.*

Proof. Since this property is orthogonal to the effects, we refer to [Wright and Felleisen 1994] for a full proof. Termination is guaranteed since we have no recursion or fixed point operator. The type and reduction rules together guarantee that a well-typed term cannot get stuck. □

Of course, full Flix includes recursion, so it allows non-terminating terms. With the operational semantics in place, we can now define soundness and completeness of the type system w.r.t. effects:

$$
\begin{array}{rcl}
(\lambda x.\, e)\, v & \xrightarrow{\mathsf{T}} & e[x := v] \\
\mathsf{let}\, x = v\, \mathsf{in}\, e & \xrightarrow{\mathsf{T}} & e[x := v] \\
\mathsf{let}\, x : \sigma = v\, \mathsf{in}\, e & \xrightarrow{\mathsf{T}} & e[x := v] \\
\mathsf{if}\, \mathsf{T}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 & \xrightarrow{\mathsf{T}} & e_1 \\
\mathsf{if}\, \mathsf{F}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 & \xrightarrow{\mathsf{T}} & e_2 \\
v; e & \xrightarrow{\mathsf{T}} & e \\
\mathsf{print}\, v & \xrightarrow{\mathsf{F}} & \mathsf{skip}
\end{array}
$$

(a) Reduction Rules.

$$
\begin{array}{rcl}
C & ::= & [\,] \mid C\, e \mid v\, C \\
& \mid & \mathsf{let}\, x = C\, \mathsf{in}\, e \mid \mathsf{let}\, x : \sigma = C\, \mathsf{in}\, e \\
& \mid & \mathsf{print}\, C \mid C; e \mid \mathsf{if}\, C\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2
\end{array}
$$

$$
\frac{e_1 \xrightarrow{\beta} e_2}{C[e_1] \xrightarrow{\beta} C[e_2]}
$$

(b) Eager Evaluation Contexts.

Fig. 4. Operational Semantics: left-to-right eager evaluation

Definition 3.8 (Soundness and Completeness). Expression e *produces an effect* if $e \longmapsto^* e' \xrightarrow{\mathsf{F}} e''$ for some e' and e''. The type system is *sound* if for all expressions, $\Gamma \vdash_b e : \tau \,\&\, \mathsf{T}$ implies that e doesn't produce an effect. The type system is *complete* if for closed expressions, $\vdash_b e : \tau \,\&\, \mathsf{F}$ implies that e produces an effect.

Example 3.9. Consider the following expressions, where S stands for String and U stands for Unit, and their (most general) types:

- $X \stackrel{\text{def}}{=} \lambda G.G(\lambda x.\mathsf{print}\, x) : ((S \xrightarrow{\mathsf{F}} U) \xrightarrow{\beta_1} \alpha_1) \xrightarrow{\beta_1} \alpha_1 \,\&\, \mathsf{T}$
- $G_1 \stackrel{\text{def}}{=} \lambda f.\mathsf{skip} : \alpha_2 \xrightarrow{\mathsf{T}} U \,\&\, \mathsf{T}$
- $G_2 \stackrel{\text{def}}{=} \lambda f.f(''hello'') : (S \xrightarrow{\beta_2} \alpha_3) \xrightarrow{\beta_2} \alpha_3 \,\&\, \mathsf{T}$

Given these typings, we get the following:

- $(XG_1) : U \,\&\, \mathsf{T}$ doesn't produce a side effect, so for this example the type system is sound:

$$XG_1 \longmapsto^{\mathsf{T}} G_1(\lambda x.\mathsf{print}\, x) \longmapsto^{\mathsf{T}} \mathsf{skip}$$

- $(XG_2) : U \,\&\, \mathsf{F}$ does produce a side effect, so for this example the type system is complete:

$$XG_2 \longmapsto^{\mathsf{T}} G_2(\lambda x.\mathsf{print}\, x) \longmapsto^{\mathsf{T}} (\lambda x.\mathsf{print}\, x)''hello'' \longmapsto^{\mathsf{T}} \mathsf{print}\, ''hello'' \xrightarrow{\mathsf{F}} \mathsf{skip}$$

We will demonstrate that the type system is sound. We will also show that the over-approximation in rule (T-Ite) is the only reason why the system is not complete. The proof follows the syntactic approach by [Wright and Felleisen 1994]. The key observation is that reduction steps preserve the type. However, in our case, the types can become "more pure" by a reduction of if-then-else. The key to proving type preservation, is the substitution lemma. It states that we can safely substitute a pure expression of the proper type into an expression. We will need this later to substitute values (which are always pure).

Lemma 3.10 (Substitution). *If* $\Gamma, x : \forall \overline{\gamma}.\tau_1 \vdash_b e_2 : \tau_2 \,\&\, \varphi$ *and* $\Gamma \vdash_b e_1 : \tau_1 \,\&\, \mathsf{T}$ *and* $\overline{\gamma} \cap ftv(\Gamma) = \emptyset$, *then* $\Gamma \vdash_b e_2[x := e_1] : \tau_2 \,\&\, \varphi$.

Proof. Induction on (the derivation of) e_2, along the lines of [Wright and Felleisen 1994].[4] □

[4]The side condition on ftv is necessary in order to avoid blocking a generalization. For instance, consider expression $e_2 \equiv \{z : \beta \to \beta, y : \forall \alpha.\alpha \to \alpha\} \vdash_b \mathsf{let}\, x = y\, \mathsf{in}\, x(0); x(\mathsf{T})$, where y must be polymorphic. We cannot substitute $[y := z : \beta \to \beta]$, since it would block the generalization of x. Another example that does not work is substituting $[y := \lambda u.u + 0 : \mathrm{Nat} \to \mathrm{Nat}]$. However, substituting any term of type $\alpha \to \alpha$, like $[y := \lambda u.u]$ works fine.

Note that the substitution lemma would get more complicated if we would substitute impure expressions $e_1 : \tau_1 \,\&\, \varphi_1$. For instance, if $e_2 \equiv \lambda y.\, e_2' : \tau_1 \xrightarrow{\varphi_2} \tau_2$, then we could have $(\lambda y.\, e_2')[x := e_1] : \tau_1 \xrightarrow{\varphi_1 \wedge \varphi_2} \tau_2$, or $(\lambda y.\, e_2')[x := e_1] : \tau_1 \xrightarrow{\varphi_2} \tau_2$, depending on whether $x \in FV(\lambda y.\, e_1)$.

The following lemma considers what happens to the type if we reduce the subject. This depends on the exact setting. In particular, if we allow if-then-else with its short-cutting (non-eager) reduction rules, we might lose effects during reduction (recall that we use $\Rightarrow$ for logical implication). We also need the more liberal (T-Seq') to obtain type preservation.

Lemma 3.11 (Subject Reduction / Type Preservation). *Let $\Gamma \vdash_b e_1 : \tau \,\&\, \varphi_1$ and $e_1 \longmapsto e_2$ then:*

(1) In the system with (T-Seq): e_2 may become untypeable.
(2) In the system with (T-Seq') and without (T-Seq), (T-Ite) and (T-Prt): $\Gamma \vdash_b e_2 : \tau \,\&\, \varphi_1$
(3) In the full system with (T-Seq') instead of (T-Seq): $\Gamma \vdash_b e_2 : \tau \,\&\, \varphi_2$ for some φ_2 with $\varphi_1 \Rightarrow \varphi_2$.

Proof.

(1) With (T-Seq) we have $(\mathsf{print}\,(s); 17); 18 : \mathsf{Nat} \,\&\, \mathsf{F}$ and $(\mathsf{print}\,(s); 17); 18 \xmapsto{\mathsf{F}} (skip; 17); 18 \xmapsto{\mathsf{T}} 17; 18$, which cannot be typed, since $17 : \mathsf{Nat} \,\&\, \mathsf{T}$ is pure.
(2) Inspection of all rewrite rules in Figure 4a and using the Substitution Lemma. It is also crucial that the contexts in Figure 4b don't allow reductions under a λ, since (T-Abs) is the (only) rule where the effect φ of a subterm is lifted to T.
(3) If $e_1 : \tau \,\&\, \varphi_1$ and $e_2 : \tau \,\&\, \varphi_2$, then $\mathsf{if}\,\mathsf{T}\,\mathsf{then}\,e_1\,\mathsf{else}\,e_2 : \tau \,\&\, \varphi_1 \wedge \varphi_2$. Indeed, $\varphi_1 \wedge \varphi_2 \Rightarrow \varphi_1$. Similar when the test is F. Finally, $\mathsf{print}\,v : \mathsf{Unit} \;\&\; \mathsf{F}$ and $skip : \mathsf{Unit} \;\&\; \mathsf{T}$ and $\mathsf{F} \Rightarrow \mathsf{T}$.

□

We conjecture that an extension of the evaluation rules to lazy evaluation (i.e., allowing non-values in the rewrite rules for λ and let), would still preserve the types in the sense of (3): in this case an effect could be lost before it is executed, similar to the short-cutting semantics of if-then-else. For example, $(\lambda x.\, 0)\,(\mathsf{print}\,s) \overset{lazy}{\longmapsto} 0$. Proving type preservation (and thus soundness) for lazy evaluation in general, however, would require a more complicated substitution lemma.

The main theorem shows that the effect system is sound, in the sense that pure terms, $e : \tau \,\&\, \mathsf{T}$ can never produce a side effect.

Theorem 3.12 (Soundness). *The type system $\vdash_b$ is sound, i.e. if $\Gamma \vdash_b e : \tau \,\&\, \mathsf{T}$, then e doesn't produce an effect.*

Proof. Assume $\Gamma \vdash_b e : \tau \,\&\, \mathsf{T}$. Note that we can replace every application of (T-Seq) by (T-Seq') and (T-Eq) (using $\mathsf{F} \wedge \varphi_2 \equiv_{\mathcal{B}} \mathsf{F}$). So the typing judgement holds in the system with (T-Seq') as well. By Subject Reduction (3), any e' with $e \longmapsto^* e'$ has $e' : \tau \,\&\, \varphi$, with $\mathsf{T} \Rightarrow \varphi$, i.e. $\varphi \equiv \mathsf{T}$. So the next step cannot be the print step, which is the only step that produces a side effect. □

Theorem 3.13 (Completeness). *The type system $\vdash_b$ without (T-Ite) is complete, i.e. if $\vdash_b e : \tau \,\&\, \mathsf{F}$, for some closed expression e without if-then-else, then e produces an effect.*

Proof. Assume $\Gamma \vdash_b e : \tau \,\&\, \mathsf{F}$. As before, we can type this in (T-Seq') as well. Since e is closed and typed, by Lemma 3.7, e reduces to a value v. By Subject Reduction (2), this reduction must contain a print-step, since otherwise we would have that $v : \tau \,\&\, \mathsf{F}$, but all values are pure. □

Of course, in full Flix with recursion, there are also impure non-terminating terms without if-then-else that never produce an effect.

3.5 Type Inference

We now show how to compute a most general type, by extending the classical type inference "Algorithm W" [Hindley 1969; Milner 1978]. Figure 5 presents the algorithm as yet another inference system. The inference rules can be interpreted as an algorithm: given context Γ and expression e, the algorithm computes its most general type τ, effect φ and also returns a substitution S instantiating some free type variables from context Γ. The conditions in each rule should be read from left-to-right. In particular, the substition computed from the first type check is typically applied on the inputs of the second type check.

The algorithm makes use of several side conditions, some with a side effect:

- α/β = freshVar(): returns a fresh type/effect variable (side effect on some global counter)
- τ = inst(σ): if $\sigma = \forall\overline{\gamma}\tau$, this returns τ with $\overline{\gamma}$ replaced by fresh variables.
- $S = (\varphi_1 \stackrel{?}{=} \varphi_2)$: this indicates a call to the Boolean unification algorithm, cf. Section 3.2.
- $S = (\tau_1 \stackrel{?}{=} \tau_2)$: this indicates a call to the type unification algorithm, explained below.
- $S = (\sigma_1 \sqsubseteq_{\mathcal{B}} \sigma_2)$: this checks if σ_1 is more liberal than σ_2, as explained below.

The first two conditions cannot fail, but the latter three can fail. If they fail, the type inference algorithm reports a type error.

The computed effects are normalized, as indicated by the notation $\varphi\!\downarrow$, according to some set of rewrite rules. This normalisation is the "syntax-directed" counter-part of the (T-Eq) rule of Figure 3.

Checking the side conditions. It is well known that pure types have a most general unifier. In [Boudet et al. 1989] it is shown that the combination of a Boolean algebra and a free algebra still has a most general unifier. Here we just sketch the procedure to check $\tau_1 \stackrel{?}{=} \tau_2$, where $X?$ is a set of flexible (type or Boolean) variables that can be instantiated. All other atomic symbols are type constants or rigid variables (Y). The first rule that matches should be applied. In Appendix A we provide a formulation in the style of Martelli and Montanari [1982] and some examples.

$$
\begin{array}{rcll}
\mathsf{unify}(\tau \stackrel{?}{=} \tau) & = & \emptyset & \\
\mathsf{unify}(x \stackrel{?}{=} \tau) & = & \mathit{FAIL} & \text{if } x \in \mathsf{ftv}(\tau) \text{ (“occur check”)} \\
\mathsf{unify}(x \stackrel{?}{=} \tau) & = & \{x \mapsto \tau\} & \text{if } x \in X? \text{ and } x \notin \mathsf{ftv}(\tau) \\
\mathsf{unify}(\tau \stackrel{?}{=} x) & = & \mathsf{unify}(x \stackrel{?}{=} \tau) & \text{if } x \in X? \text{ and } \tau \notin X? \\
\mathsf{unify}(\tau_1 \xrightarrow{\varphi_1} \tau_1' \stackrel{?}{=} \tau_2 \xrightarrow{\varphi_2} \tau_2') & = & & \text{where } S_1 = \mathsf{unify}_{\mathcal{B}}(\varphi_1 \stackrel{?}{=} \varphi_2), \\
\quad \mathsf{unify}(S_2(S_1(\tau_1')) \stackrel{?}{=} S_2(S_1(\tau_2'))) \circ S_2 \circ S_1 & & & \text{and } S_2 = \mathsf{unify}(S_1(\tau_1) \stackrel{?}{=} S_1(\tau_2)) \\
\mathsf{unify}(\tau \stackrel{?}{=} \tau') & = & \mathit{FAIL} & \text{otherwise}
\end{array}
$$

We also need a procedure to check if $\sigma_1 \sqsubseteq_{\mathcal{B}} \sigma_2$ holds, i.e. type scheme σ_1 is more liberal than σ_2. Let $\sigma_1 = \forall\gamma_1.\tau_1$ and $\sigma_2 = \forall\gamma_2.\tau_2$. By definition $\sigma_1 \sqsubseteq_{\mathcal{B}} \sigma_2$ if and only if $\forall\gamma_2.\exists\gamma_1.\tau_1 \equiv_{\mathcal{B}} \tau_2$. We check this by solving the unification problem $\mathsf{unify}(\tau_1(\gamma_1?) \stackrel{?}{=} \tau_2(\gamma_2))$, i.e. we unify the corresponding monotypes, considering the bound variables from σ_1 as flexible, and those from σ_2 (as well as all free type variables from the context) as rigid.

Properties of Algorithm W. Algorithm W yields correct results, if all inferred types lead to valid type judgements. This can be stated and proved along the lines in [Damas 1984].

Theorem 3.14 (Correctness). *If* $\Gamma \vdash_w e : \tau \,\&\, \varphi, S$ *then* $S\Gamma \vdash_b e : \tau \,\&\, \varphi$.

Proof. Induction on e and case distinction on the last $\vdash_w$ rule applied. Several cases need that type judgments, generalisation, and $\sqsubseteq_{\mathcal{B}}$ are closed under substitution. Rule (T-Eq) is required after each unification (since it only guarantees $\tau_1 \equiv_{\mathcal{B}} \tau_2$) and after each simplification (since $\varphi \equiv_{\mathcal{B}} \varphi\!\downarrow$). We refer to the technical report for the detailed proof. □

$$\frac{\mathsf{typeOf}(c) = \sigma \quad \tau = \mathsf{inst}(\sigma)}{\Gamma \vdash_w c : \tau \,\&\, \mathsf{T}, \emptyset} \text{ (W-Cst)}$$

$$\frac{(x, \sigma) \in \Gamma \quad \tau = \mathsf{inst}(\sigma)}{\Gamma \vdash_w x : \tau \,\&\, \mathsf{T}, \emptyset} \text{ (W-Var)}$$

$$\frac{\alpha = \mathsf{freshVar}() \quad \Gamma, x : \alpha \vdash_w e : \tau \,\&\, \varphi, S}{\Gamma \vdash_w \lambda x.\, e : S\alpha \xrightarrow{\varphi} \tau \,\&\, \mathsf{T}, S} \text{ (W-Abs)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_w e_1 : \tau_1 \,\&\, \varphi_1, S_1 \quad S_1\Gamma \vdash_w e_2 : \tau_2 \,\&\, \varphi_2, S_2 \\ \alpha = \mathsf{freshVar}() \quad \beta = \mathsf{freshVar}() \\ S_3 = (S_2\tau_1 \stackrel{?}{=} \tau_2 \xrightarrow{\beta} \alpha)\end{array}}{\Gamma \vdash_w e_1\, e_2 : S_3\alpha \,\&\, S_3(S_2\varphi_1 \wedge \varphi_2 \wedge \beta)\downarrow, S_3S_2S_1} \text{ (W-App)}$$

$$\frac{\Gamma \vdash_w e_1 : \tau_1 \,\&\, \varphi_1, S_1 \quad S_2 = (\varphi_1 \stackrel{?}{=} \mathsf{F}) \quad S_2S_1\Gamma \vdash_w e_2 : \tau_2 \,\&\, \varphi_2, S_3}{\Gamma \vdash_w e_1; e_2 : \tau_2 \,\&\, \mathsf{F}, S_3S_2S_1} \text{ (W-Seq)}$$

$$\frac{\Gamma \vdash_w e_1 : \tau_1 \,\&\, \varphi_1, S_1 \quad \sigma = \mathsf{gen}(S_1\Gamma, \tau_1) \quad S_1\Gamma, x : \sigma \vdash_w e_2 : \tau_2 \,\&\, \varphi_2, S_2}{\Gamma \vdash_w \mathsf{let}\, x = e_1 \,\mathsf{in}\, e_2 : \tau_2 \,\&\, (S_2\varphi_1 \wedge \varphi_2)\downarrow, S_2S_1} \text{ (W-Let-1)}$$

$$\frac{\Gamma \vdash_w e_1 : \tau_1 \,\&\, \varphi_1, S_1 \quad \mathsf{gen}(S_1\Gamma, \tau_1) \sqsubseteq_{\mathcal{B}} \sigma \quad S_1\Gamma, x : \sigma \vdash_w e_2 : \tau_2 \,\&\, \varphi_2, S_2}{\Gamma \vdash_w \mathsf{let}\, x : \sigma = e_1 \,\mathsf{in}\, e_2 : \tau_2 \,\&\, (S_2\varphi_1 \wedge \varphi_2)\downarrow, S_2S_1} \text{ (W-Let-2)}$$

$$\frac{\begin{array}{c}\Gamma \vdash_w e_1 : \tau_1 \,\&\, \varphi_1, S_0 \quad S_1 = (\tau_1 \stackrel{?}{=} \mathsf{Bool}) \\ S_1S_0\Gamma \vdash_w e_2 : \tau_2 \,\&\, \varphi_2, S_2 \\ S_2S_1S_0\Gamma \vdash_w e_3 : \tau_3 \,\&\, \varphi_3, S_3 \quad S_4 = (S_3\tau_2 \stackrel{?}{=} \tau_3)\end{array}}{\begin{array}{l}\Gamma \vdash_w \mathsf{if}\, e_1 \,\mathsf{then}\, e_2 \,\mathsf{else}\, e_3 : S_4(\tau_3) \,\& \\ S_4(S_3(S_2\varphi_1 \wedge \varphi_2) \wedge \varphi_3)\downarrow, S_4S_3S_2S_1\end{array}} \text{ (W-If)}$$

$$\frac{\Gamma \vdash_w e : \tau \,\&\, \varphi, S_0 \quad S_1 = (\tau \stackrel{?}{=} \mathsf{String})}{\Gamma \vdash_w \mathsf{print}\, e : \mathsf{Unit} \,\&\, \mathsf{F}, S_1S_0} \text{ (W-Print)}$$

Fig. 5. Type Inference Rules for λ_{eff} with judgements of the form $\Gamma \vdash_w e : \tau \,\&\, \varphi, S$.

Algorithm W is complete, if it infers the most general type and effect for every typeable expression.

Theorem 3.15 (Completeness). *If $S\Gamma \vdash_b e : \tau \,\&\, \varphi$ then for some τ', φ' and S', $\Gamma \vdash_w e : \tau' \,\&\, \varphi', S'$ and there exists S_0 such that $S_0S'\Gamma = S\Gamma$, $S_0\mathit{gen}(S'\Gamma, \tau') \sqsubseteq_{\mathcal{B}} \tau$ and $S_0\varphi' \sqsubseteq_{\mathcal{B}} \varphi$.*

Proof. Induction on e and case distinction on the last $\vdash_b$ rule applied. Several cases require that (Boolean) unification yields a most general unifier. □

Corollary 3.16 (Principal Types). *$\vdash_b$ satisfies the principal type property.*

Complexity. The worst case complexity of type inference is quite dramatic. Already for plain Hindley-Milner, typeability is DEXPTIME-complete [Mairson 1990]. To solve constraints on polymorphic effects, we add Boolean unification on top of this.

Using (T-Let-2) one can easily encode both satisfiability and validity of any formula Q, with free Boolean variables $\overline{\beta}$. Consider the expression

$$\mathsf{let}\, G : \forall\overline{\beta}.(\mathsf{Unit} \xrightarrow{Q} \mathsf{Unit}) \to \mathsf{Unit} = (\lambda f.\mathsf{skip})\,\mathsf{in}\, G(\lambda x.\,\mathsf{skip}) \ .$$

Here G is applied to the pure argument $\lambda f.\mathsf{skip}$, but expects an argument with effect Q. So the expression is well typed if and only if Q is satisfiable, which is an NP-complete problem. On the other hand, the following expression is only well-typed if Q is equivalent to T, so if Q is valid, which is a co-NP-complete problem.

$$\mathsf{let}\, H : \forall\overline{\beta}.\mathsf{Unit} \xrightarrow{Q} \mathsf{Unit} = \lambda x.\mathsf{skip}\,\mathsf{in}\,(\mathsf{if}\,\mathsf{T}\,\mathsf{then}\,H\,\mathsf{else}\,\lambda x.\mathsf{skip}) \ .$$

To check $\sigma \sqsubseteq_{\mathcal{B}} \sigma'$, we need Boolean unification with constants, which is Π_2^P-complete [Baader 1998]. Indeed, we solve problems of the form $\forall Y.\exists X.\varphi(X, Y) \equiv_{\mathcal{B}} \psi(X, Y)$, which corresponds to a general QBF_2 problem. Clearly, the overall procedure stays within the DEXPTIME-complexity of plain Hindley-Milner. Fortunately, the experimental evaluation will show that the overhead of type inference with effects stays also modest in practice, just as for plain ML.

3.6 Comparison to Row Polymorphic Effects

Hindley-Milner can be extended with row polymorphic effects [Leijen 2014, 2017]. In a row polymorphic effect system, the sequence expression:

```
Console.printLine("Hello World"); throw new RuntimeException()
```

is assigned the row: {Print, Exception}. A row polymorphic effect is sound: it over-approximates the computational effects of an expression. However it is imprecise in the following sense: the sequence expression and all of its sub-expressions:

```
123; throw new RuntimeException()
```

are assigned the effect row {Exception}. In particular, the pure expression 123 is assigned the effect row {Exception} despite the fact that this expression obviously cannot throw an exception. In contrast, our type and effect system accurately assigns purity (and impurity) to each expression.

4 IMPLEMENTATION

We have implemented the type and effect system as an extension of the Flix programming language.

4.1 The Flix Programming Language

Flix is a functional-first, imperative, and logic programming language that supports algebraic data types, pattern matching, parametric polymorphism, currying, higher-order functions, extensible records, first-class Datalog constraints, channel and process-based concurrency, and tail call elimination [Madsen and Lhoták 2018, 2020; Madsen et al. 2016]. The Flix compiler is implemented in 60,000 lines of Scala code. Flix ships with a standard library written in Flix. The standard library is approximately 10,000 lines of code and has more than 1,000 functions.

4.2 Extension with Polymorphic Effects

Extending the compiler with polymorphic effects was arduous, but not difficult. It required many careful changes to the frontend, to the typing rules, and to the type inference implementation. The integration of the successive variable elimination algorithm into Algorithm W was straightforward. In total, we added or changed approximately 5,000 lines of code. Of these, approximately 500 lines were required for the successive variable elimination algorithm and Boolean simplification rules. We have included the implementation of the most important functions in the technical report.

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

https://flix.dev/

The source code, including the source code for Boolean unification, is available at:

https://github.com/flix/flix.

5 EVALUATION

We can now state our overall empirical research question:

> Given that Boolean unification is NP–hard, is it practical to perform effect inference using Boolean constraints for a realistic programming language?

To answer this question, we consider several more specific research questions:

- **RQ1**: What is the impact on the size of the types?
- **RQ2**: What is the performance cost of effect inference?
- **RQ3**: What is the importance of simplifying Boolean formulas?
- **RQ4**: What is the performance impact on end-to-end compilation time?

Table 2. Breakdown of the core Flix standard library.

		Overview of Standard Library Functions				
			Higher-Order			
Module	Lines	Functions	HOF	Pure	Impure	Poly
Base num types	2,483	248	-	-	-	-
Prelude	286	38	16	7 (44%)	0 (0%)	9 (56%)
Array	1,475	140	73	33 (45%)	6 (8%)	34 (47%)
List	978	95	41	15 (37%)	3 (7%)	23 (56%)
Map	559	61	37	8 (22%)	3 (8%)	26 (70%)
Nel	326	36	19	7 (37%)	1 (5%)	11 (58%)
Option	383	40	22	5 (23%)	1 (5%)	16 (73%)
Result	293	30	18	4 (22%)	1 (6%)	13 (72%)
Set	405	44	19	8 (42%)	3 (16%)	8 (42%)
String	1,300	126	37	13 (35%)	5 (14%)	19 (51%)
Validation	220	20	13	2 (15%)	0 (0%)	11 (85%)
MutSet	311	35	16	8 (50%)	1 (6%)	7 (44%)
MutMap	414	48	27	8 (30%)	1 (4%)	18 (66%)
RedBlackTree	405	27	15	4 (27%)	1 (7%)	10 (66%)
StringBuilder	179	19	2	2 (100%)	0 (0%)	0 (0%)
Totals	10,017	1,007	355	124 (35%)	26 (7%)	205 (58%)

5.1 Benchmark Programs

To investigate these research questions, we consider two corpora of Flix programs: the Flix standard library and a collection of Flix applications. We answer RQ1–RQ3 with experiments based on the standard library, whereas we use both the standard library and the Flix applications to answer RQ4. We focus on the standard library for RQ1–RQ3 because it is a significant source of higher-order polymorphic functions. Library code typically contains significantly more polymorphism than application code. Consequently, library code presents a greater challenge for effect inference.

5.1.1 The Flix Standard Library. We have annotated the Flix standard library with appropriate effects as outlined in Section 2. In practice, this required us to decide when to mark higher-order functions as accepting pure, impure, or effect-polymorphic functions. Table 2 shows a breakdown of the *core* Flix standard library. The *core* standard library includes everything in the standard library, except for a few compiler intrinsics and some work-in-progress functionality, which contains very few higher-order functions. Table 2 is divided into three categories: modules that deal with primitive types (where there no higher-order functions), modules that deal with collections (where there are many higher-order functions), and miscellaneous modules (where there are some higher-order functions). The table has the following columns: The Module column is the name of the module. The Lines column is the number of lines of source code in the module. The Functions column is the total number of functions in the module. The HOF column is the total number of *Higher-Order Functions* in the module. The Pure column is the number of higher-order functions that takes an explicitly *pure* function argument. The Impure column is the number of higher-order functions that takes an explicitly *impure* function argument. The Poly column is the number of higher-order functions that takes an *effect polymorphic* function argument. The number in the parentheses is the percentage out of the number of higher-order functions. As an example, consider the Option module. The Option

module consists of 383 lines of code. The module has 40 functions of which 22 are higher-order. Out of these 22 higher-order functions, 5 (23%) require pure function arguments, 1 (5%) requires impure function arguments, and 16 (73%) accept effect polymorphic function arguments. Because of rounding, the percentages may not precisely add up to 100%.

The table demonstrates that there is a use for pure, impure, and effect polymorphic function annotations. Most higher-order functions are effect polymorphic (58%), many require pure function arguments (35%) while fewer require impure function arguments (7%).

The standard library is representative of typical idiomatic Flix code. Specifically: (i) it was written before the effect system was designed, (ii) it is inspired by the Haskell and Scala standard libraries borrowing a large part of their API surface, (iii) while many functions are relatively small, this is indicative of being well-designed rather than being trivial, and finally (iv) as mentioned earlier, the standard library is a rich source of higher-order effect polymorphic functions, which are the most challenging for type inference.

5.1.2 A Collection of Flix Applications. Table 3 shows a collection of Flix applications used to answer RQ4. All applications were written before the current work and had to be lightly updated with new effect annotations. We will give more details about the applications later.

5.2 Experimental Setup

We ran all experiments on a Windows 10 Desktop PC with an Intel Core i5-8600K CPU @ 3.60Ghz with 6 physical cores and with 16GB RAM. We used openjdk version "13" 2019-09-17 with default options. The Flix compiler was compiled with Scala 2.13.1.

We ran all the experiments on a warmed-up JVM. This is realistic because the most common use case is for the compiler to be run in a background process while serving requests to an IDE through the Language Server Protocol.

We now address each research question in turn.

5.3 RQ1: What is the impact on the size of the types?

Definition 5.1 (Type Size). We define the size of a type as the number of nodes in its abstract syntax tree.

Definition 5.2 (Substitution Size). We define the size of a substitution as the sum of the sizes of all the types in its co-domain.

We measure the size of every computed substitution for every function definition in the core standard library with and without effects. Figure 6 shows a scatter plot of substitution sizes. Each data point (x, y) consists of two measurements for a single function where x is the substitution size without effects and y is the substitution size with effects. For example, the data point Array.map = $(82, 92)$ means that the substitution size without effects is 82 and with effects it is 92 .

We expect every data point to be above the line $y = x$ since substitutions with effects are at least as large as substitutions without effects. Figure 6 confirms that this is the case. The figure shows that effects only cause a very modest increase in the size of the computed substitutions. In particular, we see no evidence of an exponential blow-up in the size of the types.

We answer RQ1 by concluding that effects only modestly increases the size of the types. The overhead appears to be between 1.0x and 1.15x.

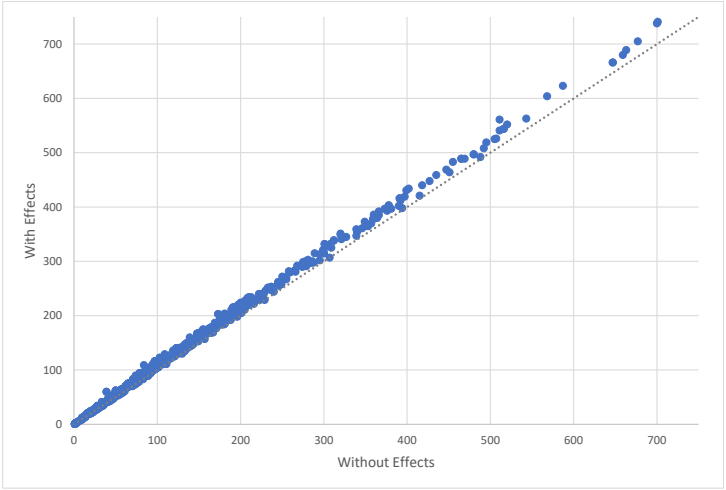

Fig. 6. A scatter plot of substitution size with and without effects.

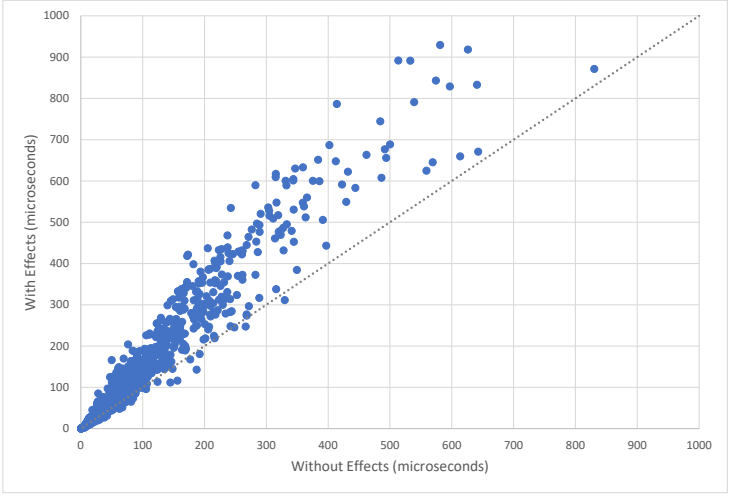

Fig. 7. A scatter plot of the time spent on inference with and without effects.

5.4 RQ2: What is the performance cost of effect inference?

We measure the time spent in type (and effect) inference for each function in the core standard library with and without effects. Specifically, we run the inference algorithm for each function 1,000 times and take the median of the measured execution times.

Figure 7 shows a scatter plot of the execution time with and without effects. The horizontal axis shows the running time, without effects, measured in microseconds. The vertical axis show the running time, with effects, measured in microseconds. Each data point (x, y) consists of two measurements for a single function where x is the time without effects and y is the time with effects.

For example, the data point Array.map = $(62, 126)$ means that type inference without effects took $62\mu s$ and type inference with effects took $126\mu s$.

We expect every data point to be above the line $y = x$ since type inference without effects should be at least as fast as type inference with effects. Figure 7 confirms that this is the case except for a few outliers that may be due to imprecision in the measurements. The figure shows no evidence of an exponential blow-up in type inference time. The results suggest a performance cost of around 1.0x to 2.0x slowdown. The execution time appears predictable and stable; there are no extreme outliers. When we measure the average overhead of the entire type and effect inference phase, we arrive at a slowdown of 1.4x (the average overhead is computed as the ratio between the sum total of time spent on type inference without effects vs. the sum total of time spent on type inference with effect inference.)

We answer RQ2 by concluding that the performance impact of effect inference with Boolean unification is between a 1.0x to 2.0x slowdown with an average slowdown of 1.4x. We see no evidence of any exponential blow-up.

5.5 RQ3: What is the importance of simplifying Boolean formulas?

To understand the performance impact of simplifying Boolean formulas inside the successive variable elimination algorithm, we disable all simplification rules, except those that are required to evaluate closed terms. We measure the time spent in type and effect inference, as in RQ2.

We very quickly discover that without simplifications the SVE algorithm is completely impractical: The compiler runs out of memory when trying to compile almost any program. Without simplification, the Boolean formulas grow very quickly due to the nature of the SVE algorithm.

We answer RQ3 by concluding that Boolean simplifications are *absolutely necessary* to make the SVE algorithm practical for type and effect inference. The simplifications used by the implementation are available in the technical report.

5.6 RQ4: What is the performance impact on end-to-end compilation time?

To understand the performance impact of effect inference with Boolean unification on end-to-end compilation time we must first understand the structure of the Flix compiler. Figure 8 shows a breakdown of the time spent in each phase of the compiler without effect inference when compiling the standard library together with the library tests. The figure shows that 51.5% of compilation time is spent on JVM bytecode generation, 12.1% on type inference, 9.0% on monomorphization,

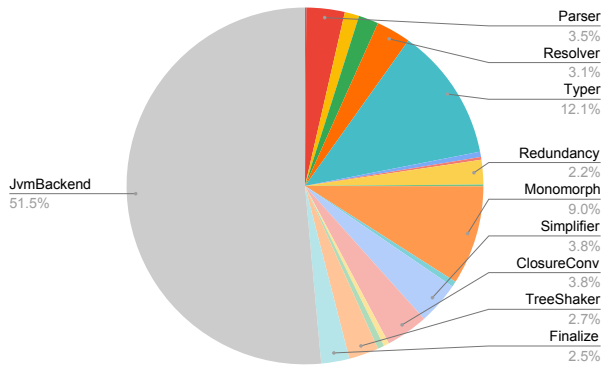

Fig. 8. A breakdown of time spent in each phase of the Flix compiler (without effect inference).

Table 3. Compiler throughput with and without effects.

Program	Lines	Throughput (lines/sec) No Effects	With Effects	Slowdown
Core Standard Library	10,017	82,984	75,879	1.09x
Standard Library	10,901	89,661	77,881	1.15x
Compiler Tests + (Standard Library)	21,294	57,108	54,148	1.05x
Library Tests + (Standard Library)	25,556	18,414	16,670	1.10x
Abstract Domains + (Standard Library)	13,652	45,338	45,605	0.99x
Puppet Master + (Standard Library)	11,754	58,104	51,228	1.13x
Flix Time + (Standard Library)	13,835	83,579	77,252	1.08x

and the rest on various less time-consuming phases. In other words, type inference is a significant part of compilation time, but it is not the most expensive compiler phase. From RQ2, we know that the performance cost of effect inference is between a 1.0x and 2.0x slowdown (average 1.4x). This in itself would increase the overall time by only 5%, but the later phases may also be affected by the larger effect types.

We measure the compiler throughput, i.e. lines of code compiled per second, with and without effect inference, for a collection of Flix programs. We measure throughput, as opposed to absolute running times, since such numbers are mutually comparable and comparable to other compilers.

We used all larger Flix programs available at the time of writing. The programs are: The Core Standard Library used in the earlier experiments. The Standard Library which is the Core Standard Library plus 900 lines of additional code. The Compiler Tests and Library Tests. The Abstract Domains library which is a collection of abstract domains implemented for the work in [Madsen and Lhoták 2018]. The Puppet Master library which is an actor framework written in Flix. The Flix Time library which exposes Java date and time utilities to Flix. The standard library is included in all of the above programs, because they depend on it and because Flix is a whole-program optimizing compiler; there is no separate compilation. The reported lines of code include the standard library.

Table 3 shows the results of this experiment. The Program column shows the name of the compiled program. The Lines column shows the total number of lines of source code in this program. The two columns, No Effects and With Effects, measure the throughput of the compiler in lines of code per second. Finally, the Slowdown column contains the ratio of the two throughputs.

The experimental results suggest that effect inference imposes an overall slowdown of between 1.0x and 1.15x on end-to-end compilation. Library Tests (including the standard library), with more than 25,000 lines of code, forms the largest of all programs and exhibits a slowdown of around 1.10x. While we cannot separate program code from standard library code, we can conclude that the program code does not reveal any exponential behavior. This is to be expected: effect inference is most expensive in the presence of higher-order effect polymorphic functions, but such functions are much more prevalent in the standard library than in any other type of code. In particular, the programs considered have significantly fewer higher-order functions, and indeed most functions are simply pure or impure.[5]

[5]The reader may wonder why the throughput of the compiler varies so significantly. For example, the standard library alone compiles much faster (77,000 lines/second) than the standard library with its tests (16,000 lines/second). The answer is simple and can be understood with reference to Figure 8. When the standard library is compiled, *without any tests*, the program does not contain any entry points and after type and effect inference aggressive tree shaking means that there is no JVM bytecode to generate. The backend of the compiler effectively becomes a no-op. In contrast, when tests are included, every test is an entry point, and the backend must now emit JVM bytecode for everything.

We answer RQ4 by concluding that the performance impact of effect inference on end-to-end compilation time is between 1.0x and 1.15x. The average slowdown is around 1.1x.

5.7 Overall Research Goal

Based on the experimental results, we conclude that type and effect inference with Boolean unification is both practical and usable. We estimate an inference overhead in the range from 1.0x to 2.0x with an estimated 1.4x for real-world programs. In terms of end-to-end compilation time, the overall overhead is 1.05x to 1.15x with an average of around 1.1x.

6 RELATED WORK

6.1 Boolean Unification

The earliest work on Boolean unification and the successive variable elimination algorithm goes back to George Boole himself [Boole 1847]. Later work includes that of Rudeanu [1974] and Buttner and Simonis [1987]. Martin and Nipkow [1989] provide an accessible introduction to Boolean unification, while Boudet et al. [1989] study unification in combinations of theories, including Boolean rings. Baader [1998] studies the computational complexity of Boolean unification in detail. Finally, Löwenheim [1908] provides an alternative algorithm for Boolean unification.

Macii et al. [1998] studies the performance Boole's algorithm (the successive variable elimination algorithm) versus Löwenheim's algorithm. The experiments, performed on a collection of circuit design problems, suggest that both methods are comparable in running time and memory use. Both implementations use binary decision diagrams (BDDs), whereas our implementation is based on explicit substitutions. We leave it as interesting future work to determine whether: (i) Löwenheim's algorithm and/or (ii) a BDD based representation of either algorithm is better suited for the implementation of type and effect inference algorithms.

Early practical applications of Boolean unification include circuit design and extensions to logic programming [Bockmayr 1991; Buttner and Simonis 1987; Simonis and Dincbas 1987]. We can now add an additional practical application: polymorphic type and effect systems.

6.2 Type Systems and Type Inference

The Damas–Hindley–Milner type system was discovered by Hindley [1969] and Milner [1978]. Later, Damas studied the formal foundations for the type system in his PhD thesis [Damas 1984]. The Damas–Hindley–Milner type system has been used as the foundation for several popular functional programming languages, including Standard ML, OCaml, and Haskell. Over the years, a plethora of extensions to the Damas–Hindley–Milner type system have been proposed, including support for qualified-types [Jones 2003], type classes [Wadler and Blott 1989], region-based memory management [Tofte and Talpin 1997], polymorphic mutable references [Tofte 1990], and more.

In an influential work, Wright and Felleisen [1994] describe a "syntactic approach" to proving soundness of type systems, including the Damas–Hindley–Milner type system. We extend their proof strategy, based on the subject reduction theorem, with polymorphic effects.

Vytiniotis et al. [2010] argue against generalization of local let-bindings for real-world programming languages. The authors argue that local-let generalization: (i) is rarely used in practice, and (ii) significantly complicates the implementation of type inference. In this paper, we have presented the meta-theory of the type and effect system in the style of Hindley-Milner, but the Flix extension follows the advice of Vytiniotis et al.

6.3 Early Type and Effect Systems with Inference

Rytz et al. [2012] present a light-weight effect polymorphic type system for the Scala programming language. The type system supports two notions of effectful functions: A function $f : T \overset{e}{\Rightarrow} U$ has effect e when evaluated whereas a function $g : T \xrightarrow{e} U$ has *both* the effect of e *and* the effect of any higher-order function in its argument T. That is, the function type of g *implicitly* includes the effect of its argument. The type system supports sub-effecting through a lattice of effects embedded in the type hierarchy of the language. In our type system there is only one function type of the form: $a \xrightarrow{e} b$ were e is the effect of the function. Every function is effect polymorphic by default, but the programmer may explicitly require a pure or impure function by using the arrows -> and ~> which are simply syntactic sugar for $a \xrightarrow{\mathsf{T}} b$ and $a \xrightarrow{\mathsf{F}} b$, respectively. While our system has only a single effect (purity), effects can be combined with Boolean operations.

Leroy and Pessaux [2000] present a type-based analysis of uncaught exceptions for programming languages in the ML-family. The "static analysis" presented in the paper is formulated as a row polymorphic type and effect system and uses type inference to compute the set of exceptions that an expression may throw. The authors report on the performance impact of their type and effect inference; the results suggest that "On average, the exception analysis takes twice as much time as OCaml type inference." The authors describe these performance results as "quite good". While our type and effect system is very different from theirs, our experimental results suggest an average type inference overhead in the same range.

Toro and Tanter [2015] present a gradual effect system for Scala. Gradual typing allows the programmer to mix static and dynamic-typing: type and effect annotations are provided for some parts of the program. Type safety is then guaranteed through a mix of compile-time and run-time checks. At a glance, gradual typing and our work may look similar: the programmer writes certain type and effect annotations while omitting others. However, type inference and gradual typing are fundamentally different. In our system type safety is *always* guaranteed at compile-time; the type and effect annotations are merely used to further constrain the set of typeable programs.

6.4 Algebraic Effect Systems

Algebraic effects were initially introduced by [Plotkin and Pretnar 2009]. Algebraic effects combine dynamic scoping and multi-prompt delimited continuations [Asai and Kameyama 2007; Danvy and Filinski 1990] enabling the programmer to trigger and handle effects. In programming languages with algebraic effects, the programmer may invoke a (possibly user-defined) effect. Then the run-time captures the current (delimited) continuation, finds the nearest effect handler, and invokes it with the continuation. This mechanism naturally generalizes typical exception and exception handling mechanisms by allowing access to the current (delimited) continuation.

In recent years many new programming languages with algebraic effect systems have been proposed, including Eff [Bauer and Pretnar 2015], Koka [Leijen 2005], Frank [Lindley et al. 2017], and Effekt [Schuster et al. 2020]. In addition, several (embedded) domain specific programming languages (eDSLs) for algebraic effects have also been proposed, including libraries for Idris [Brady 2013] and Scala [Brachthäuser et al. 2020]. Efforts have also been directed towards extending existing programming languages with algebraic effects, notably for OCaml [Dolan et al. 2015; Kiselyov and Sivaramakrishnan 2018].

A major topic of research on algebraic effects is the design of type and effect systems that capture the nature of triggering and handling effects. For example, to ensure *effect safety* – that all effects of an expression are handled. In relation to our work, most closely related are algebraic effects systems that support type and effect inference. A common technique is based on *row polymorphism* as exemplified by the Koka programming language [Leijen 2005].

6.5 Row Polymorphism

A row polymorphic type and effect system infers a row (a list) of effects for every expression. In this paper, we have focused on a type and effect system with a judgement $\Gamma \vdash e : \varphi$ that focuses on a single effect φ which captures when e is pure. While we have not yet done so, it seems clear that the typing judgement can easily be extended to $\Gamma \vdash e : \varphi_1, \cdots, \varphi_n$ to describe multiple effects $\varphi_1, \cdots, \varphi_n$. On the other hand, our type and effect system with a single effect is more fine-grained and expressive than a row polymorphic system. Our system is *more fine-grained* since each expression is assigned a precise effect, whereas in a row polymorphic system each expression is assigned an over-approximate effect row. Consequently, a row polymorphic effect system may assign a pure expression an effect. Our system is *more expressive*, since we can specify higher-order functions that require e.g. at least one pure function or at most one impure function argument, etc. We have found such types useful for expressing programming lints and rewrite rules. Such constraints cannot be expressed with row polymorphism.

6.6 Using Purity to Ease the Value Restriction

The value restriction is a well-known rule to limit generalization in the presence of mutability to ensure type safety [Garrigue 2004; Talpin and Jouvelot 1994; Wright 1995]. Under standard Hindley-Milner type rules, if we have the let-binding:

```
1  let id = x -> x;
2  (id(123), id("hello world"))
```

the `id` variable is given the polymorphic type $\forall\alpha.\, \alpha \to \alpha$, and the program type checks. In the presence of mutability, e.g. reference cells or arrays, unrestricted generalization is unsound:

```
1  let r = ref None;
2  let r1: Option[Bool] = r;
3  let r2: Option[Int32] = r;
4  r1 := Some(true);
5  Option.map(x -> x + 1, (deref r2))
```

Here `r` is given the polymorphic type $\forall\alpha.$ Option$[\alpha]$ and the program type checks. However, if the program is evaluated it gets stuck because the lambda `x -> x + 1` cannot be applied to a Boolean. The value restriction overcomes this problem by requiring that only *syntactic values*[6] are generalized. Thus the above program does not type check, but neither does the program:

```
1  let id = x -> x;     // id can generalized because it is a syntactic value,
2  let id2 = id(id);    // whereas id2 cannot be generalized.
3  (id2(123), id2("hello world"))
```

which is now unfairly rejected by the type checker. We can use our polymorphic effect system to ease the value restriction: if an expression is pure then it cannot allocate, read, nor write mutable memory, and hence it is safe to generalize.

7 FUTURE WORK

We briefly describe two interesting directions for future work. Flix, like any programming language that targets the JVM and aims at interoperability with Java, must deal with at least two thorny issues without compromising safety: null values and exceptions.

[6] http://mlton.org/ValueRestriction

Null. Famously dubbed the "billion dollar mistake" by its inventor Sir Tony Hoare, the `null` value is an endless source of `NullPointerExceptions` in Java programs. Flix does not support `null` and instead recommends the use of the safer `Option` type. However, Flix still wants access to the huge Java ecosystem and so some kind of `null`-support is still required. In future work, we want to explore whether Boolean constraints can be used to safely type `null`. We imagine a typing judgement $\Gamma \vdash e : \tau \,?\, \eta$ where, for every expression, we know the type of the expression and whether it may evaluate to `null`. If η is `F` then the expression *cannot* evaluate to `null`. If η is `T` then the expression may evaluate to `null` and an explicit null-check is required. A function may be polymorphic in its nullity, for example, we might imagine a map function with the signature:

```
1  def map[a, b, n](f: a -> b ? n, l: List[a]): List[b ? n] = ...
```

where f and l must be a non-`null` and the elements of l must be a non-`null`. The function is polymorphic in the *nullity* of the result of f. That is, if f returns non-`null` values then the resulting list has non-`null` elements. If, on the other hand, f returns `null` values then the resulting list may have `null` values. Using the richer language of Boolean constraints, it should be possible to express even more interesting function types, for instance, a polymorphic function that returns a non-`null` value when given two arguments of which at least one is non-`null`.

Exceptions. Flix does not support exceptions and instead recommends the use of the `Result` type. Given that Java has exceptions, some kind of exception support is still required for interoperability. We imagine an approach, similar to that for `null`, for capturing whether an expression may throw an exception. Standard exception mechanisms allow the programmer to selectively catch and handle certain exceptions while letting other exceptions propagate down the call stack. We are not yet sure how to model such selective exception behavior. On the positive side, it seems more clear that we can support support higher-order functions that are polymorphic in their exception behavior, and we can support fine-grained reasoning about the exception behavior of individual expressions.

We believe that our effect system could also be used for fine-grained reasoning about information flow security, blocking and non-blocking I/O, and control operators such as shift/reset.

8 CONCLUSION

We have presented a simple, practical, and expressive type and effect system based on Boolean constraints. The effect system extends the Hindley-Milner type system, supports parametric polymorphism, and preserves principal types modulo Boolean equivalence. We have shown how to extend Algorithm W with Boolean unification based on the successive variable elimination algorithm. In this paper, we have used the type and effect system to track and enforce purity. In future work, we plan to extend the work to model nullity and exceptions.

We have implemented the type and effect system as an extension of the type system in the Flix programming language. We have also updated the Flix standard library with appropriate effect annotations. The extension is freely available, open source, and ready for use.

The worst case complexity of Boolean unification is NP-hard (even Π_2^P-complete with constants), which is "negligible" in view of the worst-case DEXPTIME-completeness of ML type inference.

Fortunately, this also holds up in practice. Like for plain Hindley-Milner, our approach to add polymorphic effects appears to be practical for a real-world programming language. Experimental evaluation on the Flix standard library and Flix applications have shown that: (i) the impact of polymorphic effects on type inference time is a 1.4x slowdown, and (ii) the overall impact on end-to-end compilation time is a 1.1x slowdown. Based on these experimental results, we believe our approach is both practical and useful.

A DETAILS ON BOOLEAN UNIFICATION

A.1 Boolean Unification

We illustrate the SVE algorithm with an example. We will write + instead of xor.

Consider the unification problem $a \wedge b \stackrel{?}{=} c \wedge d$, from Table 1.(10). We will eliminate variables a and b, in this order, which appears to be sufficient if we simplify intermediate formulas.

Define $\varphi := a \wedge b + c \wedge d$. We use SVE to solve the matching problem $\varphi \stackrel{?}{=} \mathsf{F}$. Replacing a by F and T, respectively, yields

$$\begin{array}{lcl} \varphi_0 & = & c \wedge d \\ \varphi_1 & = & b + c \wedge d \end{array}$$

The next recursive call to SVE is on their conjunction, which is equivalent to $\psi := c \wedge d \wedge \neg b$. Replacing now b by F and T, respectively, yields:

$$\begin{array}{lcl} \psi_0 & = & c \wedge d \\ \psi_1 & = & \mathsf{F} \end{array}$$

Their conjunction is F, so the next SVE call succeeds, and returns the empty substitution. We can now compute the solution for b. It is the substitution

$$S_1 : b \mapsto \psi_0 \vee (b \wedge \neg\psi_1) \equiv (c \wedge d) \vee (b \wedge \neg\mathsf{F}) \equiv (c \wedge d) \vee b$$

Finally, we can compute the solution for a, which is

$$a \mapsto S_1(\varphi_0) \vee (a \wedge \neg S_1(\varphi_1)) \equiv (c \wedge d) \vee (a \wedge \neg(((c \wedge d) \vee b) + (c \wedge d))) \equiv (c \wedge d) \vee (a \wedge \neg b)$$

So we have computed the most general unifier $\{b \mapsto (c \wedge d) \vee b \ ; \ a \mapsto (c \wedge d) \vee (a \wedge \neg b)\}$.

A.2 Type and Effect Unification Algorithm

Next, we show how to combine the Martelli-Montanari unification algorithm with Boolean Unification. The system is presented as a set of rewrite rules that should be applied to a set of equations E until it reaches a fixpoint. Note that mgus for Boolean constraints are not necessarily idempotent, but we can always assume w.l.o.g. that we obtain an idempotent mgu, i.e. we assume that if $(\beta = \psi) \in sve(\varphi)$, then some renaming has been applied, such that $\beta \notin \mathrm{ftv}(\psi)$.

(U-Refl)	$\{\tau \stackrel{?}{=} \tau\} \cup E \mapsto E$	
(U-Symm)	$\{\tau \stackrel{?}{=} \alpha\} \cup E \mapsto \{\alpha \stackrel{?}{=} \tau\} \cup E$	if $\tau \notin TypeVar$
(U-Occur)	$\{\alpha \stackrel{?}{=} \tau\} \cup E \mapsto \bot$	if $\alpha \in \mathrm{ftv}(\tau)$ and $\alpha \neq \tau$
(U-Subst)	$\{\alpha \stackrel{?}{=} \tau\} \cup E \mapsto \{\alpha = \tau\} \cup [\alpha \mapsto \tau](E)$	if $\alpha \notin \mathrm{ftv}(\tau)$
(U-Arrow)	$\{\tau_1 \xrightarrow{\varphi_1} \tau_2 \stackrel{?}{=} \tau_3 \xrightarrow{\varphi_2} \tau_4\} \cup E \mapsto \{\tau_1 \stackrel{?}{=} \tau_3, \tau_2 \stackrel{?}{=} \tau_4, \varphi_1 \stackrel{?}{=} \varphi_2\} \cup E$	
(U-BSucc)	$\{\varphi_1 \stackrel{?}{=} \varphi_2\} \cup E \mapsto S \cup S(E)$	if $S = sve(\varphi_1 \stackrel{?}{=} \varphi_2)$
(U-BFail)	$\{\varphi_1 \stackrel{?}{=} \varphi_2\} \cup E \mapsto \bot$	if $sve(\varphi_1 \stackrel{?}{=} \varphi_2)$ fails
(U-Simpl)	$\{\varphi_1 \stackrel{?}{=} \varphi_2\} \cup E \mapsto \{\varphi_1' \stackrel{?}{=} \varphi_2'\} \cup E$	if $\varphi_i' = \mathrm{simplify}(\varphi_i)$, where $\mathrm{simplify}(\varphi)$ simplifies the Boolean formula φ.

We now give two examples of how the algorithm works. We use blue text to indicate the equation we simplify in each step.

A.3 Example I

We start with the unification problem:

$$\{(\tau_1 \xrightarrow{\beta} \tau_2) \xrightarrow{\beta} \tau_3 \stackrel{?}{=} (\tau_1 \xrightarrow{\gamma} \tau_2) \xrightarrow{\neg\gamma} \tau_3\}$$

By (U-Arrow), we obtain:

$$\{(\tau_1 \xrightarrow{\beta} \tau_2) \stackrel{?}{=} (\tau_1 \xrightarrow{\gamma} \tau_2), \tau_3 \stackrel{?}{=} \tau_3, \beta \stackrel{?}{=} \neg\gamma\}$$

By (U-Refl), we obtain:

$$\{(\tau_1 \xrightarrow{\beta} \tau_2) \stackrel{?}{=} (\tau_1 \xrightarrow{\gamma} \tau_2), \beta \stackrel{?}{=} \neg\gamma\}$$

By (U-BSucc), assuming that $\beta \stackrel{?}{=} \neg\gamma$ yields $\gamma \mapsto \neg\beta$, we obtain:

$$\{(\tau_1 \xrightarrow{\beta} \tau_2) \stackrel{?}{=} (\tau_1 \xrightarrow{\neg\beta} \tau_2), \gamma = \neg\beta\}$$

By (U-Arrow), we obtain:

$$\{\tau_1 \stackrel{?}{=} \tau_1, \tau_2 \stackrel{?}{=} \tau_2, \beta \stackrel{?}{=} \neg\beta, \gamma = \neg\beta\}$$

By (U-BFail), we obtain:

$$\bot$$

A.4 Example II

We start with the unification problem:

$$\{(\tau \xrightarrow{x\wedge y} \tau) \xrightarrow{x\vee y} \tau \stackrel{?}{=} (\tau \xrightarrow{F} \tau) \xrightarrow{T} \tau\}$$

By (U-Arrow), we obtain:

$$\{\tau \xrightarrow{x\wedge y} \tau \stackrel{?}{=} \tau \xrightarrow{F} \tau, \tau \stackrel{?}{=} \tau, x \vee y \stackrel{?}{=} T\}$$

By (U-Refl), we obtain:

$$\{\tau \xrightarrow{x\wedge y} \tau \stackrel{?}{=} \tau \xrightarrow{F} \tau, x \vee y \stackrel{?}{=} T\}$$

By (U-BSucc), assuming that $x \vee y \stackrel{?}{=} T$ yields $x \to v \vee \neg u, y \to u$, we obtain:

$$\{\tau \xrightarrow{(v\vee\neg u)\wedge u} \tau \stackrel{?}{=} \tau \xrightarrow{F} \tau, x = v \vee \neg u, y = u\}$$

By (U-Simpl), we obtain:

$$\{\tau \xrightarrow{v\wedge u} \tau \stackrel{?}{=} \tau \xrightarrow{F} \tau, x = v \vee \neg u, y = u\}$$

By (U-Arrow), we obtain:

$$\{\tau \stackrel{?}{=} \tau, \tau \stackrel{?}{=} \tau, v \wedge u \stackrel{?}{=} F, x = v \vee \neg u, y = u\}$$

By (U-Refl, twice), we obtain:

$$\{v \wedge u \stackrel{?}{=} F, x = v \vee \neg u, y = u\}$$

By (U-BSucc), assuming that $v \wedge u \stackrel{?}{=} F$ yields $v \to s \wedge \neg t, u \to t$, we obtain:

$$\{v = s \wedge \neg t, u = t, x = (s \wedge \neg t) \vee \neg t, y = t\}$$

By (U-Simpl), and removing intermediate variables, we obtain:

$$\{x = \neg t, y = t\}$$

and we are done.

If we apply this mgu M to the type, we can calculate:

$$M((\tau \xrightarrow{x\wedge y} \tau) \xrightarrow{x\vee y} \tau)$$
$$(\tau \xrightarrow{\neg t\wedge t} \tau) \xrightarrow{\neg t\vee t} \tau$$
$$(\tau \xrightarrow{F} \tau) \xrightarrow{T} \tau$$

as was expected.

REFERENCES

Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic delimited continuations. In *Asian Symposium on Programming Languages and Systems*.

Franz Baader. 1998. On the complexity of Boolean unification. *Inform. Process. Lett.* (1998).

Andrej Bauer and Matija Pretnar. 2015. Programming with algebraic effects and handlers. *Journal of logical and algebraic methods in programming* (2015).

Alexander Bockmayr. 1991. Logic Programming with Pseudo-Boolean Constraints. (1991).

George Boole. 1847. *The mathematical analysis of logic.*

Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauß. 1989. Unification in Boolean Rings and Abelian Groups. *Journal of Symbolic Computation* (1989). https://doi.org/10.1016/S0747-7171(89)80054-9

Jonathan Immanuel Brachthäuser, Philipp Schuster, and Klaus Ostermann. 2020. Effekt: Capability-passing style for type-and effect-safe, extensible effect handlers in Scala. *Journal of Functional Programming* (2020).

Edwin Brady. 2013. Programming and reasoning with algebraic effects and dependent types. In *Proc. International Conference on Functional Programming (ICFP)*.

Wolfram Buttner and Helmut Simonis. 1987. Embedding Boolean expressions into logic programming. *Journal of Symbolic Computation* (1987).

Luis Damas. 1984. *Type assignment in programming languages.* Ph.D. Dissertation. The University of Edinburgh.

Olivier Danvy and Andrzej Filinski. 1990. Abstracting control. In *Proc. Conference on LISP and Functional Programming*.

Stephen Dolan, Leo White, KC Sivaramakrishnan, Jeremy Yallop, and Anil Madhavapeddy. 2015. Effective concurrency through algebraic effects. In *The OCaml Workshop*.

Jacques Garrigue. 2004. Relaxing the value restriction. In *International Symposium on Functional and Logic Programming*.

Roger Hindley. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society (AMS)* (1969).

Mark P Jones. 2003. *Qualified types: theory and practice.* Cambridge University Press.

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *The Haskell Workshop*.

Oleg Kiselyov and KC Sivaramakrishnan. 2018. Eff directly in OCaml. *arXiv preprint arXiv:1812.11664* (2018).

Daan Leijen. 2005. Extensible records with scoped labels. *Trends in Functional Programming (TFP)* (2005).

Daan Leijen. 2014. Koka: Programming with row polymorphic effect types. *Workshop on Mathematically Structured Functional Programming (MSFP)* (2014).

Daan Leijen. 2017. Type directed compilation of row-typed algebraic effects. In *Proc. Symposium on Principles of Programming Languages (POPL)*.

Xavier Leroy and François Pessaux. 2000. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (2000).

Sam Lindley, Conor McBride, and Craig McLaughlin. 2017. Do Be Do Be Do. In *Proc. Principles of Programming Languages (POPL)*.

Leopold Löwenheim. 1908. *Über das Auflösungsproblem im logischen Klassenkalkul.*

John M Lucassen and David K Gifford. 1988. Polymorphic effect systems. In *Proc. Principles of Programming Languages (POPL)*.

Enrico Macii, Giuseppe Odasso, and Massimo Poncino. 1998. Comparing different Boolean unification algorithms. In *Proc. Asilomar Conference on Signals, Systems and Computer*.

Magnus Madsen and Ondřej Lhoták. 2018. Safe and sound program analysis with Flix. In *Proc. International Symposium on Software Testing and Analysis (ISSTA)*.

Magnus Madsen and Ondřej Lhoták. 2020. Fixpoints for the Masses: Programming with First-class Datalog Constraints. *Proc. ACM Programming Languages* Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) (2020).

Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: a declarative language for fixed points on lattices. In *Proc. Programming Language Design and Implementation (PLDI)*.

Harry G. Mairson. 1990. Deciding ML Typability is Complete for Deterministic Exponential Time. In *Proc. Principles of Programming Languages (POPL)*.

Alberto Martelli and Ugo Montanari. 1982. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)* (1982).

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

Robin Milner. 1978. A theory of type polymorphism in programming. *J. Comput. System Sci.* (1978).

Gordon Plotkin and Matija Pretnar. 2009. Handlers of algebraic effects. In *European Symposium on Programming (ESOP)*.

Sergiu Rudeanu. 1974. *Boolean functions and equations.*

Lukas Rytz, Martin Odersky, and Philipp Haller. 2012. Lightweight polymorphic effects. In *Proc. European Conference on Object-Oriented Programming (ECOOP)*.

Philipp Schuster, Jonathan Immanuel Brachthäuser, and Klaus Ostermann. 2020. Compiling effect handlers in capability-passing style. *Proc. International Conference on Functional Programming (ICFP)* (2020).

Helmut Simonis and Mehmet Dincbas. 1987. Using an extended Prolog for digital circuit design. In *IEEE International Workshop on AI Applications to CAD Systems for Electronics*.

Jean-Pierre Talpin and Pierre Jouvelot. 1994. The type and effect discipline. *Information and Computation* (1994).

Mads Tofte. 1990. Type inference for polymorphic references. *Information and Computation* (1990).

Mads Tofte and Jean-Pierre Talpin. 1997. Region-based memory management. *Information and Computation* (1997).

Matías Toro and Éric Tanter. 2015. Customizable gradual polymorphic effects for Scala. *Proc. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2015).

Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. 2010. Let should not be generalized. In *Proc. Workshop on Types in Language Design and Implementation*.

Philip Wadler and Stephen Blott. 1989. How to make ad-hoc polymorphism less ad hoc. In *Proc. Symposium on Principles of Programming Languages (POPL)*.

Andrew K Wright. 1995. Simple imperative polymorphism. *Lisp and Symbolic Computation* (1995).

Andrew K Wright and Matthias Felleisen. 1994. A syntactic approach to type soundness. *Information and Computation* (1994).

Yichen Xie and Dawson Engler. 2002. Using redundancies to find errors. In *Proc. Symposium on Foundations of Software Engineering (FSE)*.