Restrictable Variants: A Simple and Practical Alternative to Extensible Variants

Magnus Madsen
Department of Computer Science, Aarhus University, Denmark

Jonathan Lindegaard Starup
Department of Computer Science, Aarhus University, Denmark

Matthew Lutze
Department of Computer Science, Aarhus University, Denmark

Abstract

We propose restrictable variants as a simple and practical alternative to extensible variants. Restrictable variants combine nominal and structural typing: a restrictable variant is an algebraic data type indexed by a type-level set formula that captures its set of active labels. We introduce new pattern-matching constructs that allows programmers to write functions that only match on a subset of variants, i.e., pattern-matches may be non-exhaustive. We then present a type system for restrictable variants which ensures that such non-exhaustive matches cannot get stuck at runtime.

An essential feature of restrictable variants is that the type system can capture structure-preserving transformations: specifically the introduction and elimination of variants. This property is important for writing reusable functions, yet many row-based extensible variant systems lack it.

In this paper, we present a calculus with restrictable variants, two partial pattern-matching constructs, and a type system that ensures progress and preservation. The type system extends Hindley-Milner with restrictable variants and supports type inference with an extension of Algorithm W with Boolean unification. We implement restrictable variants as an extension of the Flix programming language and conduct a few case studies to illustrate their practical usefulness.

2012 ACM Subject Classification Theory of computation → Program semantics

Keywords and phrases restrictable variants, extensible variants, refinement types, Boolean unification

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2023.17

Supplementary Material *Software (ECOOP 2023 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.9.2.12`

1 Introduction

> "Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away." – *Antoine de Saint-Exupéry*

In functional programming, algebraic data types and pattern matching have been hugely successful. So successful that many non-functional mainstream programming languages, including Kotlin and Rust have also adopted them. While algebraic data types, i.e. sum and variant types, are widely used, their cousins *extensible variants* and *extensible records* are far less available. Extensible variants and records, based on row-polymorphic type systems, have been known for several decades [11, 16, 37]. Yet one has to look far to find usable implementations. OCaml does not support extensible records, but does support a form of extensible variants as a "language extension", but this implementation is far less powerful than simple row-polymorphic systems. PureScript supports extensible records, but not extensible variants.[1] Elm had support for extensible records, but this feature was removed.[2] We speculate that there are at least a few reasons for this lack of support: (i) lack of real (or perceived) use cases, (ii) implementation difficulty, and (iii) hitting the "right" expressiveness.

[1] `https://github.com/purescript/documentation/blob/master/language/Records.md`
[2] `https://github.com/elm/compiler/issues/985`

To expand on (iii), we believe we have identified a major practical weakness in existing row-based extensible variant systems. We illustrate the problem with an example: In a compiler pipeline, we can view each compiler phase as a function, and the whole compiler as the composition of these functions. For example, we might have:

```
let compile = ... >> typecheck >> lambdalift >> codegen
```

where >> is forward function composition. The lambdalift phase performs closure conversion and lambda lifting which is required before we can generate machine code. Concretely, we can imagine that the lambdalift function replaces Lambda expressions with Closure expressions in the abstract syntax tree. We say that lambdalift *introduces* the Closure variant and *eliminates* the Lambda variant. Importantly, we must run lambdalift before we can run codegen.

As compiler writers, it would be very useful if we could type check the abstract syntax trees produced by lambdalift. That is, we would like to write:

```
let compile = ... >> typecheck >> lambdalift >> typecheck >> codegen
```

This requires us to extend the typecheck function to handle Closure expressions, but that is simple; type checking them is similar to type checking lambdas.

We might think that the above scenario can be programmed with row-based extensible variants, but, unfortunately, this is not the case. The problem is the following: The codegen phase *cannot* handle abstract syntax trees unless they have been closure-converted and lambda-lifted, i.e. unless the Lambda expression has been eliminated. But after the second call to typecheck, a row-based system loses the knowledge that the Lambda variant has been eliminated, hence the above program does not type check.

We call this phenomenon the *co-domain problem* for extensible variants:

> **The Co-Domain Problem:** Type systems with extensible variants based on row polymorphism are unable to precisely capture the *introduction* and *elimination* of variants in pattern-matches. (We expand on the details in Section 5.)

To overcome this issue, we propose *restrictable variants*. A restrictable variant is a sum type indexed by a type-level set formula that over-approximates the "active" set of labels of the sum. We can think of a restrictable variant as a form of refinement type [10, 36] where the type-level index refines the possible labels of an expression of that type. In this way, restrictable variants combine nominal and structural typing. With restrictable variants, programmers can write *one* data type definition and reuse it in different contexts. This is in contrast to the standard functional programming approach of writing multiple, but similar, data types definitions or using a purely structural type system.

In this paper, we introduce restrictable variants and a new partial pattern-matching construct which comes in two flavors: `choose` and `choose*`. The `choose` expression permits a non-exhaustive pattern-match on a restrictable variant where only some variants are handled. The `choose*` expression goes further and enables programmers to write structure-preserving transformations which are captured at the type level. Specifically, we can precisely capture the *introduction* or *elimination* of variants. This overcomes the co-domain problem that was outlined above. We propose a type system for restrictable variants which is an extension of Hindley-Milner, supports complete type inference, and ensures that programs with partial pattern-matches (i.e. with `choose` and `choose*`) cannot get stuck.

We compare the expressiveness of restrictable variants to other existing systems, including row-based extensible variants [11, 16, 33, 37], row theories [30], occurrence typing [7], and relational nullable types [25]. We find that many of these systems are significantly more expressive (and sometimes more complex) than restrictable variants, yet most cannot express

the simple programming patterns that we use in our case studies. We think that restrictable variants (like row-based systems) have simple types that will be understandable by ordinary programmers and which will work well in practice. It is also our hope that restrictable variants can serve as inspiration for new and more sophisticated type systems that can handle the use cases we present.

The ideas in this paper are simple, but as far as we can tell, they have not yet been explored in the literature, and we believe they solve real problems. While we present *restrictable variants* as an alternative to *extensible variants*, we have found that it is natural to combine restrictable variants with *extensible records* (a point we return to in Section 6).

We implement restrictable variants as an extension of the Flix programming language. We discuss how the implementation supports complete type inference as a natural extension of Algorithm W. The two key ideas are: (i) a type rule, for the `choose*` expression, which relates the type-level index of the "input" (scrutinee expression) to the "output" (result expression), and (ii) a formulation of the type rule as a *set equation* which is solvable by Boolean unification in the algebra of sets.

We use the implementation to conduct a case study of a few programs that use restrictable variants. The first case study models Boolean formulas and is used as a running example throughout the paper. The second case study combines the Option, List, and Nel (non-empty list) data types into one restrictable variant. The third case study shows how to combine restrictable variants with extensible records. The case studies demonstrate that programming with restrictable variants is simple and valuable.

In summary, the contributions of this paper are:

- **(Restrictable Variants)** We present *restrictable variants*: a simple alternative to extensible variants. Restrictable variants offer a new point in the design space with different trade-offs from existing systems. Moreover, restrictable variants solve the function composition problem for row-based extensible variants.
- **(Type System)** We present a type system for restrictable variants. We prove the standard progress and preservation theorems. The type system ensures that a program with partial pattern-matches (the `choose` and `choose*` expressions) cannot get stuck.
- **(Implementation)** We implement restrictable variants as an extension of the Flix programming language. We discuss how the type system supports type inference via an extension of Algorithm W with Boolean unification.
- **(Expressiveness)** We compare the expressiveness of restrictable variants to other systems. We observe that restrictable variants are simple, yet they support reasonable use cases that cannot be handled by many other systems, in particular those based on row polymorphism.
- **(Case Study)** We present a case study of a few programs that use restrictable variants. The case study shows that (a) restrictable variants are useful and (b) capture real-world programming patterns.

This paper is organized as follows: In Section 2 we present restrictable variants and motivate their use with several examples. In Section 3 we present a type system for restrictable variants based on type-level set formulas. In Section 4 we discuss our implementation of restrictable variants and show how to implement type inference. In Section 5 we compare the expressiveness of restrictable variants to other systems, including row-based extensible variants. In Section 6 we present a case study on the use of restrictable variants. In Section 7 we present related work and in Section 8 we conclude the paper.

2 Motivation

We motivate restrictable variants with several examples. We begin with a simple example to build intuition. Next, we move on to a more realistic example of modeling Boolean formulas, which we use as a running example throughout the rest of the paper. We show many types, but, of course, the point is that they can be inferred. All examples are runnable in our extension of Flix.

▶ **Example 1 (Restrictable Variant).** We can define a *restrictable variant* data type:

```
enum Color[s] {
    case Red
    case Green
    case Blue
}
```

The Color type is *indexed* by a type variable s that ranges over the *labels* of the algebraic data type. The labels of the Color data type are: Red, Green, and Blue. The index is a set formula that captures which variants of the data type may be present. For example:

$$\mathsf{Color}[\{\}] = \emptyset \qquad \mathsf{Color}[\{\mathsf{Red}, \mathsf{Blue}\}] = \{\mathsf{Red}, \mathsf{Blue}\} \qquad \mathsf{Color}[\mathsf{Green}^{\mathtt{C}}] = \{\mathsf{Red}, \mathsf{Blue}\}$$

We can also have richer indices where a free variable is involved. For example:

$$\mathsf{Color}[s - \mathsf{Red}] \subseteq \{\mathsf{Green}, \mathsf{Blue}\} \quad \{\mathsf{Green}, \mathsf{Blue}\} \subseteq \mathsf{Color}[(s - \mathsf{Red}) + (s^{\mathtt{C}})] \subseteq \{\mathsf{Red}, \mathsf{Green}, \mathsf{Blue}\}$$

where s is a free variable. In full generality, the index is a *type-level set formula* whose valuations capture which variants of the data type may be present. As the examples show, there are many equivalent set formulas. For example, $\mathsf{Green}^{\mathtt{C}}$ is equivalent to the set $\{\mathsf{Red}, \mathsf{Blue}\}$. The set formulas may also contain variables, a fact that becomes important when we consider pattern-matches on restrictable variants.

We can write a function that only operates on some colors of a restrictable variant:

```
def isWarm(c: Color[{Red, Blue}]): Bool = choose c {
    case Red  => true
    case Blue => false
}
```

Here the type of isWarm, which can be fully inferred, captures that the function can accept any color which is either Red or Blue. Specifically, the type system ensures that it is a compile-time type error to call isWarm with the color Green.

This example demonstrates a key feature of the proposed type system: We can write pattern-matches that are non-exhaustive and have the type system ensure that a function like isWarm is never invoked with a value that is not handled.

With some intuition in place, we now move on to our running example: a restrictable variant that models Boolean formulas. We use Boolean formulas since they are well-known and they are sufficient to illustrate several key features of our system. We want to stress that the following ideas scale to more complex data types, e.g. abstract syntax trees, as we shall discuss in Section 6.

A remark on notation: In Flix source code we shall write `~S` for $S^{\mathtt{C}}$, `S1 + S2` for $S_1 \cup S_2$, and `S1 & S2` for $S_1 \cap S_2$. We use these symbols because they are in ASCII and are reminiscent of the bitwise operators. In the formal treatment of the calculus and its type system (Section 3), we will use the standard math symbols.

▶ **Example 2 (Variant Restriction).** We can define a restrictable variant for Boolean formulas:

```
enum Expr[s] {
    case Var(Int32)
    case Cst(Bool)
    case Not(Expr[s])
    case Or(Expr[s], Expr[s])
    case And(Expr[s], Expr[s])
    case Xor(Expr[s], Expr[s])
}
```

We can write a function which reduces *closed* terms to a Boolean constant:

```
def eval(e: Expr[~Var]): Bool =
    choose e {
        // Var case omitted: We can only evaluate closed terms.
        case Cst(b)      => b
        case Not(x)      => not eval(x)
        case Or(x, y)    => eval(x) or eval(y)
        case And(x, y)   => eval(x) and eval(y)
        case Xor(x, y)   => eval(x) != eval(y)
    }
```

The evaluator itself is straightforward. We simply pattern-match on each case and implement the semantics directly. What is interesting is that we cannot evaluate an open term (i.e. a formula with variables in it), hence we simply omit the Var case from the pattern-match. The type system then infers that the eval function can passed any Boolean expression as long as it does not use the Var variant. This is captured by the type Expr[{~Var}] which is equivalent to Expr[{Cst, Not, Or, And, Xor}].

▶ **Example 3 (Variant Elimination).** We can also write a Boolean formula simplifier which eliminates the Xor term by translation:

```
def simplify(e: Expr[s]): Expr[~Xor] =
    choose e {
        case Var(x)      => Var(x)
        case Cst(b)      => Cst(b)
        case Not(x)      => Not(simplify(x))
        case Or(x, y)    => Or(simplify(x), simplify(y))
        case And(x, y)   => And(simplify(x), simplify(y))
        case Xor(x, y)   =>
            let x1 = simplify(x);
            let y1 = simplify(y);
            Or(And(x1, Not(y1)), And(Not(x1), y1))
    }
```

The simplifier is also straightforward. The return type of the simplify function now excludes the possibility that the returned value can contain a Xor variant. This is captured by the type Expr[{~Xor}] which is equivalent to Expr[{Var, Cst, Not, Or, And].

The unfortunate weakness of the simplifier is that if we know that the input cannot contain any variables (e.g. the Var variant) then this information is lost in the output. For example, if we have a closed Boolean formula, *we cannot simplify it and then evaluate it* because the return type of simplify includes the Var variant in its type. We lost the knowledge that the term was closed!

The fundamental issue is that in simplify we have lost the relation between the type-level index in the argument type (i.e., Expr[s]) and the result type (i.e., Expr[~Xor]). To overcome this, we introduce the `choose*` construct. The `choose*` construct allows us to maintain a relation between the input type and the output type, as the following example shows:

▶ **Example 4 (Structure-Preserving Map).** We can use the `choose*` construct to write a structure-preserving map function:

```
def map(f: Int32 -> Int32, e: Expr[s]): Expr[s] =
    choose* e {
        case Var(x)     => Var(f(x))
        case Cst(b)     => Cst(b)
        case Not(x)     => Not(map(f, x))
        case Or(x, y)   => Or(map(f, x), map(f, y))
        case And(x, y)  => And(map(f, x), map(f, y))
        case Xor(x, y)  => Xor(map(f, x), map(f, y))
    }
```

The map function applies a function f : Int32 → Int32 to every variable in the given expression. What is essential is that the argument type is Expr[s] and the result type is Expr[s] which means that information about the "active" variants in the input is preserved in the output.

▶ **Example 5 (Simplify – Revisited).** Recall that the original version of simplify used `choose` and had the signature:

```
def simplify(e: Expr[s]): Expr[~Xor] = ...
```

If we change the implementation to use `choose*` we instead get the more precise signature:

```
def simplify(e: Expr[s]): Expr[(s - Xor) + {Not, And, Or}]
```

which captures that simplify will return an expression that may contain the Not, Or, And variants plus the Cst and Var variants, if the input contains them. We might have hoped the return type would simply be Expr[(s - Xor)], but the type system cannot exclude the Not, Or, And variants because they are introduced by elimination of Xor. Fortunately, the signature of simplify is strong enough to capture the two important properties we care about:

- The Var variant can only occur in the output if it occurs in the input.
- The Xor variant is eliminated, i.e. cannot occur in the output.

Consequently, if the input is a closed formula (i.e. lacks the Var variant) then after simplification *it will still be closed* and we can evaluate it.

With the updated simplify, we can write a function:

```
let run = simplify >> eval
```

which is inferred to have the type Expr[s − Var] → Bool where the closedness requirement is propagated "backwards" through the (forward) function composition operator >>.

▶ **Example 6 (Substitution).** We can also write a substitution function that replaces each variable in a Boolean formula with a value from an environment:

```
def subst(m: Map[Int32, Bool], e: Expr[s]): Expr[(s - Var) + Cst] =
    choose* e {
        case Var(x)     => Cst(Map.getWithDefault(x, false, m))
        case Cst(b)     => Cst(b)
        case Not(x)     => Not(subst(m, x))
        case Or(x, y)   => Or(subst(m, x), subst(m, y))
        case And(x, y)  => And(subst(m, x), subst(m, y))
        case Xor(x, y)  => Xor(subst(m, x), subst(m, y))
    }
```

We define the subst function to operate on all Boolean expressions. The return type of subst is the same as the input type (*sans* Var), but may potentially contain the Cst variant. The reason is that the type system is not sufficiently expressive to capture that the Cst variant can only occur if *either* the Var *or* the Cst variants are present in the input. This "loss of precision" only affects the Cst variant. For example, we still know that if the input cannot contain the Xor variant then neither can the output.

▶ **Example 7 (Function Composition).** Imagine that we have a fast evaluator, but it only supports the Cst, Not, And, and Or variants. We can capture this with the signature:

```
def fasteval(e: Expr[s & {Cst, Not, And, Or}]): Bool = ...
```

We can *compose* the simplify, subst, and fasteval functions as follows:

```
let fastrun = m -> simplify >> subst(m) >> fasteval
```

The (inferred) type of fastrun is:

$$\mathsf{fastrun} : \forall s.\, \mathsf{Map[Int32, Bool]} \to \mathsf{Expr[s]} \to \mathsf{Bool}$$

i.e., given an environment and a Boolean expression it computes a primitive Bool.

What is essential is that the function types of simplify, subst, and fasteval *compose* in a way that preserves the information that simplify eliminates the Xor variant and subst eliminates the Var variant, hence the final call to fasteval is valid. Looking at the types:

$$\begin{aligned}\mathsf{simplify} &: \forall s_1.\, \mathsf{Expr}[s_1] \to \mathsf{Expr}[s_1 \text{ - } \mathsf{Xor} + \{\mathsf{Not}, \mathsf{And}, \mathsf{Or}\}]\\ \mathsf{subst}(m) &: \forall s_2.\, \mathsf{Expr}[s_2] \to \mathsf{Expr}[(s_2 \text{ - } \mathsf{Var}) + \mathsf{Cst}]\\ \mathsf{fasteval} &: \forall s_3.\, \mathsf{Expr}[s_3 \,\&\, \{\mathsf{Cst}, \mathsf{Not}, \mathsf{And}, \mathsf{Or}\}] \to \mathsf{Bool}\end{aligned}$$

We see that when we apply the output of simplify as the input to subst(m), we get the type:

$$\mathsf{Expr}[(((s_1 \text{ - } \mathsf{Xor} + \{\mathsf{Not}, \mathsf{And}, \mathsf{Or}\})) \text{ - } \mathsf{Var}) + \mathsf{Cst}]$$

This type is compatible with the input type of fasteval because the *set equation*:

$$(((s_1 \text{ - } \mathsf{Xor} + \{\mathsf{Not}, \mathsf{And}, \mathsf{Or}\})) \text{ - } \mathsf{Var}) + \mathsf{Cst} = s_3 \,\&\, \{\mathsf{Cst}, \mathsf{Not}, \mathsf{And}, \mathsf{Or}\}$$

has a solution. Specifically, it has the most-general unifier:

$$\{s_3 \mapsto s_1 + \{\mathsf{Cst}, \mathsf{Not}, \mathsf{And}, \mathsf{Or}\}\}$$

where s_1 and s_2 are implicitly mapped to themselves. This solution can be found by Boolean unification. Thus, in summary, we are able to infer that fastrun has the type $\forall s.\, \mathsf{Map[Int32, Bool]} \to \mathsf{Expr[s]} \to \mathsf{Bool}$ which means that it works for any Boolean formula.

As we shall discuss in Section 5, the power of our system is this ability to track the *introduction* and *elimination* of variants *through function composition*. Notably, several other existing systems lack this property, including row-based extensible variants. The key issue is that a row polymorphic system is unable to precisely relate the input type of a (partial) pattern-match to its output type. Thus we lose track of the fact that simplify eliminates the Xor variant and hence we cannot call fasteval. We call this phenomenon the "co-domain" problem for extensible variants since these type systems lack the ability to relate the domain of a (partial) pattern-match (i.e. its input type) to its co-domain (i.e. its output type).

In our experience and based on the case studies (Section 6), we find it important to stress how important this property is for reusability. In a compiler, we want to write the subst function *once* and for the *entire* abstract syntax tree. However, if the subst function loses information about what variants can be returned in its output, then its utility is hampered, as most compiler phases only operate on a subset of the entire abstract syntax tree.

2.1 Summary

We conclude with a summary of the properties of the proposed system:

- **(Property I)** Restrictable variants are sum types indexed by a type-level set formula that over-approximates the "active" set of labels of the sum. Programmers can use restrictable variants to write *one* data type definition that is reusable in many different contexts.
- **(Property II)** The `choose` construct enables programmers to write non-exhaustive pattern-matches on restrictable variants handling only the relevant cases. The `choose*` construct enables a form of refinement typing where the result type of a non-exhaustive pattern-match is related to its input type.
- **(Property III)** Functions on restrictable variants *compose* under introduction and elimination of labels; i.e., a sequence of introductions and eliminations does not lose information at the type level.
- **(Property IV)** The type system ensures that the non-exhaustive `choose` or `choose*` constructs cannot get stuck at runtime. The type system extends Hindley-Milner and supports complete type inference.
- **(Property V)** Restrictable variants are a natural generalization of algebraic data types and are simple to implement.

3 Restrictable Variants

We now present $\lambda^{\mathsf{res}}_{\mathsf{var}}$: a minimal lambda calculus with restrictable variants. We present its syntax and semantics, then its type system, and finally its meta theoretic properties. The $\lambda^{\mathsf{res}}_{\mathsf{var}}$ calculus and its type system are mostly standard; the novelties are the `choose` and `choose*` constructs and the use of *set formulas* in the type system.

3.1 Syntax and Semantics

We begin with a discussion of the syntax and semantics of the $\lambda^{\mathsf{res}}_{\mathsf{var}}$ calculus.

Syntax

The syntax of the $\lambda^{\mathsf{res}}_{\mathsf{var}}$ calculus (cf. Figure 1a) includes the standard lambda calculus constructs: variables, constants, lambda abstractions, and function applications. The let-expression allows polymorphic generalization, as is standard in Hindley-Milner. We include the if-then-else expression to illustrate how the type system merges information. We require that every $\lambda^{\mathsf{res}}_{\mathsf{var}}$ program comes with a map $\Sigma : \mathit{Enum} \to \mathit{Label} \to \mathit{Scheme}$ of declared variants.

The raison d'être is the $\mathsf{choose}\, e\, \{\overline{\eta}\}$ and $\mathsf{choose}\star\, e\, \{\overline{\eta}\}$ expressions. In both expressions, e is the match expression and $\overline{\eta}$ is a sequence of match cases. A match case is of the form $\mathsf{case}\, \mathcal{E}.\ell(x) \Rightarrow e$ where $\mathcal{E}$ is the enum that the label ℓ belongs to, x is the match variable, and e is the match expression body. As shown, we prefix all labels with the enum they come from; i.e., we write "Color.Red" and not just "Red". Recall that both choose expressions are needed, since `choose` allows expression bodies to have an arbitrary type, whereas `choose*` requires that the expression bodies have the same type as the match expression (modulo the type-level

$$
\begin{array}{rcl}
v \in \mathit{Val} & = & () \mid \texttt{true} \mid \texttt{false} \\
& \mid & \lambda x.\, e \\
& \mid & \mathcal{E}.\ell(v) \\
e \in \mathit{Exp} & = & x \mid v \mid e\, e \mid \mathcal{E}.\ell(e) \\
& \mid & \mathsf{let}\, x = e\, \mathsf{in}\, e \\
& \mid & \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \\
& \mid & \mathsf{choose}\, e\, \{\overline{\eta}\} \\
& \mid & \mathsf{choose}\star\, e\, \{\overline{\eta}\} \\
& \mid & \mathsf{open}\, e \\
\eta \in \mathit{Case} & = & \mathsf{case}\, \mathcal{E}.\ell(x) \Rightarrow e \\
\\
\mathcal{E} \in \mathit{Enum} & = & \text{a set of enums} \\
\ell \in \mathit{Tag} & = & \text{a set of tags} \\
x, y \in \mathit{Var} & = & \text{a set of variables}
\end{array}
$$

(a) Syntax of $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

$$
\begin{array}{rcl}
\varphi \in \mathit{Formula} & = & \emptyset \mid \{\mathcal{E}.\ell\} \mid \beta \mid \varphi^{\complement} \mid \varphi \cup \varphi \mid \varphi \cap \varphi \\
\tau \in \mathit{Type} & = & \alpha \mid \mathsf{Unit} \mid \mathsf{Bool} \mid \tau \to \tau \mid \mathcal{E}[\varphi] \\
\\
\sigma \in \mathit{Scheme} & = & \tau \mid \forall \alpha.\, \sigma \mid \forall \beta.\, \sigma \\
\\
\alpha \in \mathit{TypeVar} & = & \text{a set of type variables} \\
\beta \in \mathit{BoolVar} & = & \text{a set of Boolean variables}
\end{array}
$$

(b) Types and Type Schemes of $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

Figure 1 Syntax and Types of $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

$$
\begin{array}{rr}
(\lambda x.\, e)\, v \rightsquigarrow e[x \mapsto v] & \text{(E-App)} \\
\mathsf{let}\, x = v\, \mathsf{in}\, e \rightsquigarrow e[x \mapsto v] & \text{(E-Let)} \\
\mathsf{if}\, \texttt{true}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightsquigarrow e_1 & \text{(E-Ite-T)} \\
\mathsf{if}\, \texttt{false}\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2 \rightsquigarrow e_2 & \text{(E-Ite-F)} \\
\\
\mathsf{open}\, \mathcal{E}.\ell(v) \rightsquigarrow \mathcal{E}.\ell(v) & \text{(E-Open)} \\
\\
\dfrac{\eta_i = \mathsf{case}\, \mathcal{E}.\ell(x) \Rightarrow e}{\mathsf{choose}\, \mathcal{E}.\ell(v)\, \{\overline{\eta}\} \rightsquigarrow e[x \mapsto v]} & \text{(E-Choose)} \\
\\
\dfrac{\eta_i = \mathsf{case}\, \mathcal{E}.\ell(x) \Rightarrow e}{\mathsf{choose}\star\, \mathcal{E}.\ell(v)\, \{\overline{\eta}\} \rightsquigarrow \mathsf{open}\, e[x \mapsto v]} & \text{(E-Choose-}\star\text{)}
\end{array}
$$

$$
\begin{array}{rr}
\dfrac{e_1 \rightsquigarrow e_1'}{e_1\, e_2 \rightsquigarrow e_1'\, e_2} & \text{(C-App)} \\
\\
\dfrac{e \rightsquigarrow e'}{v\, e \rightsquigarrow v\, e} & \text{(C-App2)} \\
\\
\dfrac{e_1 \rightsquigarrow e_1'}{\mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 \rightsquigarrow \mathsf{let}\, x = e_1'\, \mathsf{in}\, e_2} & \text{(C-Let)} \\
\\
\dfrac{e_1 \rightsquigarrow e_1'}{\mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 \rightsquigarrow \mathsf{if}\, e_1'\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3} & \text{(C-Ite)} \\
\\
\dfrac{e \rightsquigarrow e'}{\mathsf{open}\, e \rightsquigarrow \mathsf{open}\, e'} & \text{(C-Open)} \\
\\
\dfrac{e \rightsquigarrow e'}{\mathsf{choose}\, e\, \{\overline{\eta}\} \rightsquigarrow \mathsf{choose}\, e'\, \{\overline{\eta}\}} & \text{(C-Choose)} \\
\\
\dfrac{e \rightsquigarrow e'}{\mathsf{choose}\star\, e\, \{\overline{\eta}\} \rightsquigarrow \mathsf{choose}\star\, e'\, \{\overline{\eta}\}} & \text{(C-Choose-}\star\text{)}
\end{array}
$$

Figure 2 Evaluation Rules of $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

indices). We construct a variant value with the $\mathcal{E}.\ell(e)$ expression, e.g. "Color.Red()" where () is the unit value. The $\mathsf{choose}\, e\, \{\overline{\eta}\}$ and $\mathsf{choose}\star\, e\, \{\overline{\eta}\}$ expressions are akin to pattern-matches, except there are no wildcards, tuple patterns, or nested patterns. Importantly, the `choose` and `choose*` expressions do not have to be exhaustive.

The $\mathsf{open}\, e$ expression is not part of the surface syntax, and is present only during evaluation. Semantically, $\mathsf{open}\, e$ is equivalent to e. Its purpose is explained in Section 3.2.

Semantics

The semantics of $\lambda^{\mathsf{res}}_{\mathsf{var}}$ is a call-by-value operational semantics for the lambda calculus. Figure 2 shows the evaluation rules of $\lambda^{\mathsf{res}}_{\mathsf{var}}$ which are standard except for (E-Open),(E-Choose), and (E-Choose-$\star$). We write $e[x \mapsto v]$ for the capture avoiding substitution of $x \mapsto v$ into e. The congruence rules, prefixed with C, enforce a left-to-right evaluation order. The (E-Open) rule reduces a tagged value $\mathcal{E}.\ell(v)$ to itself. The (E-Choose) evaluation rule captures that if we evaluate a tagged value $\mathcal{E}.\ell(v)$ for some value v then we look for a case $\mathsf{case}\, \mathcal{E}.\ell(x) \Rightarrow e$ in the pattern-match. If found, we evaluate the case body, i.e. we step to $e[x \mapsto v]$. The (E-Choose-$\star$) is very similar, but it instead steps to $\mathsf{open}\, e[x \mapsto v]$.

How $\lambda^{\mathsf{res}}_{\mathsf{var}}$ Programs "Get Stuck"

We briefly illustrate how $\lambda^{\mathsf{res}}_{\mathsf{var}}$ programs may get stuck during evaluation. The obvious reason is when true or false is applied as a function, or when a lambda expression is used as a condition in an if-then-else. The more interesting case is when a `choose` or `choose*` expression is applied to a variant for which there is no case:

```
choose Green {
    case Red  => true
    case Blue => false
}
```

The type system will reject such programs.

3.2 Type System

We now describe the type system of $\lambda^{\mathsf{res}}_{\mathsf{var}}$: its types, type rules, and meta-theory.

Mono Types and Poly Types (Type Schemes)

The types of $\lambda^{\mathsf{res}}_{\mathsf{var}}$ are separated into mono types (τ) and type schemes (σ). The mono types include type variables α, the base types Unit and Bool, function types $\tau \to \tau$, and variant types $\mathcal{E}[\varphi]$ which consist of an enum symbol $\mathcal{E}$ indexed by a type-level Boolean set formula φ. The language of formulas, for a given variant type $\mathcal{E}$, consists of the empty set $\emptyset$, a singleton set with one label $\{\mathcal{E}.\ell\}$, Boolean variables β, the complement of a formula $\varphi^{\complement}$, the union of two formulas $\varphi \cup \varphi$, and the intersection of two formulas $\varphi \cap \varphi$. We write $A - B$ for set difference which is equivalent to $A \cap B^{\complement}$. We also write $A <: B$ as an alias for the constraint $A - B = \emptyset$ (i.e. $A <: B \Leftrightarrow A \cap B^{\complement} = \emptyset$). Note that the complement of a set is well-defined, since a variant type is declared to have a fixed finite set of labels (which forms the universe).

In principle, to ensure that it always clear what the universe of labels is, we should always index each set formula with its associated variant type $\mathcal{E}$ symbol, e.g. we should write $\varphi_{\mathcal{E}}$. However, we typically omit the enum name when it is clear from the context.

We write $\mathsf{ftv}(\varphi)$ for the variables in φ. A *valuation* ν for a formula φ is an assignment of concrete sets of labels to all of the variables in $\mathsf{ftv}(\varphi)$. In this way, we can view a set formula as a function from concrete sets to a concrete set. Two set formulas φ_1 and φ_2 are *equivalent* (written $\varphi_1 \equiv_{\mathbb{B}} \varphi_2$) if they describe the same function. That is, if $\forall \nu.\, \nu(\varphi_1) = \nu(\varphi_2)$ where ν must be a valuation of both φ_1 and φ_2.

Type schemes σ extend types by quantification over type variables α and Boolean variables β. That is, a type scheme is of the form $\forall \overline{\gamma}.\, \tau$, where $\overline{\gamma}$ is a vector of type variables and Boolean variables. Figure 1b shows the types and type schemes of $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

We define type equivalence as the smallest relation $\equiv_{\mathbb{B}}$[3] such that:

- $\tau \equiv_{\mathbb{B}} \tau$.
- If $\tau_1 \equiv_{\mathbb{B}} \tau_1'$ and $\tau_2 \equiv_{\mathbb{B}} \tau_2'$ then $\tau_1 \to \tau_2 \equiv_{\mathbb{B}} \tau_1' \to \tau_2'$.
- If $\varphi \equiv_{\mathbb{B}} \varphi'$, then $\mathcal{E}[\varphi] \equiv_{\mathbb{B}} \mathcal{E}[\varphi']$.

For example, we have that $\mathsf{Color}[\{\mathsf{Red}\}^{\complement}] \equiv_{\mathbb{B}} \mathsf{Color}[\{\mathsf{Green}, \mathsf{Blue}\}]$. Two types, with set formulas in them, do not have to share the same variables (or even share the same number of variables) to be equivalent. For example: $\mathsf{Color}[\{\mathsf{Green}\}] \equiv_{\mathbb{B}} \mathsf{Color}[(\beta \cap \{\mathsf{Green}\}) \cup \{\mathsf{Green}\}]$.

We define substitutions $S : (\mathit{TypeVar} \cup \mathit{BoolVar} \to \mathit{Type})$ as assignment of type variables to types and Boolean variables to Boolean formulas. We say the type τ is an instance of type scheme σ, written $\sigma \sqsubseteq \tau$, if $\sigma = \forall \overline{\gamma}.\tau'$ and there exists a type substitution S such

[3] We overload the $\equiv_{\mathbb{B}}$ symbol to stand for both Boolean equivalence and type equivalence.

that $\text{dom}(S) = \overline{\gamma}$ and $S(\tau') = \tau$. Moreover, we define a context Γ as a map of bindings $x : \sigma$, and $\mathsf{ftv}(\sigma)$ to be the type variables that occur free in σ, and $\mathsf{ftv}(\Gamma)$ as the union of all free type variables in its range. We also define the generalization of a type $\mathsf{gen}(\Gamma, \tau)$ as $\forall\alpha_1, \cdots, \forall\alpha_n.\forall\beta_1, \cdots, \forall\beta_n.\tau$ where $\{\alpha_1, \cdots, \alpha_n, \beta_1, \cdots, \beta_n\} = \mathsf{ftv}(\tau) \setminus \mathsf{ftv}(\Gamma)$.

Variant Declarations

As stated earlier, we require every restrictable variant to be declared. Specifically, we assume that there is a set of enum symbols *Enum* and map $\Sigma : \mathit{Enum} \to \mathit{Label} \to \mathit{Scheme}$[4] which assigns a type scheme to every constructor (label) of the type. We require that the type schemes are of one of the two following forms[5]:

1. $\Sigma(\mathcal{E}.\ell) = \forall\beta.\, \tau \to \mathcal{E}[\beta] \qquad \mathsf{ftv}(\tau) = \emptyset$
2. $\Sigma(\mathcal{E}.\ell) = \forall\beta.\, \mathcal{E}[\beta] \to \mathcal{E}[\beta]$

These requirements ensure that:

- A constructor is applied to a simple type (e.g. $\Sigma(\mathsf{Color.Red}) = \forall\beta.\mathsf{Unit} \to \mathsf{Color}[\beta]$), or
- A constructor is applied to the same variant type, but with the *same* type-level index (e.g. $\Sigma(\mathsf{Expr.Not}) = \forall\beta.\mathsf{Expr}[\beta] \to \mathsf{Expr}[\beta]$).
- The type scheme of a constructor is always polymorphic function type over β whose result type is of the form $\mathcal{E}[\beta]$.

And that the following lemma holds:

▶ **Lemma 8** (Label-instantiation). *If two instantiations of the same label type scheme share the same result type then they must share the same argument type.*

$$\Sigma(\mathcal{E}.\ell) \sqsubseteq \tau_1 \to \mathcal{E}[\varphi] \quad \wedge \quad \Sigma(\mathcal{E}.\ell) \sqsubseteq \tau_2 \to \mathcal{E}[\varphi] \quad \Longrightarrow \quad \tau_1 = \tau_2$$

Intuitively, the result type of an instantiated label type scheme uniquely determines its argument type. The idea is that if we know that $\mathsf{Not}(e) : \mathsf{Expr}[\{\mathsf{Cst}, \mathsf{Not}\}]$ then know that the type of e is also $\mathsf{Expr}[\{\mathsf{Cst}, \mathsf{Not}\}]$. In other words, the type-level index of a restrictable variant also applies to its constituents. This fact is used to show preservation.

Type Rules

Figure 3 shows the declarative type rules of $\lambda^{\mathsf{res}}_{\mathsf{var}}$. A declarative typing judgment is of the form $\Gamma \vdash e : \tau$. As is standard, the context $\Gamma : \mathit{Var} \hookrightarrow \mathit{Scheme}$ is a partial function from variables to type schemes. Most of the type rules are standard.

The (T-Eq) rule states that if an expression e can be typed as τ_1, that type can be replaced by any equivalent type $\tau_2 \equiv_{\mathbb{B}} \tau_1$. The (T-Var) rule is the standard Hindley-Milner instantiation rule. It states that if the assumption $x : \sigma$ is in the context, then we can *instantiate* σ to a specific type τ, and conclude $x : \tau$. The (T-Let) rule is the standard Hindley-Milner generalization rule. The rule states that if we can type e_1 as τ_1 under the environment Γ then we may *generalize* the type τ_1 to a type scheme σ, and type e_2 under an extended environment with $x : \sigma$.

The (T-Tag) rule states that we can type a tag expression $\mathcal{E}.\ell(e)$ with the type $\mathcal{E}[\varphi \cup \{\mathcal{E}.\ell\}]$ where the type-level formula φ is obtained by instantiating the type scheme associated with the label $\mathcal{E}.\ell$ to $\tau \to \mathcal{E}[\varphi \cup \{\mathcal{E}.\ell\}]$. The reason that the label is not part of the scheme is that we do not want to assume the occurrence of the label when the scheme is used in (T-Choose) and (T-Choose-$\star$).

[4] In the implementation the Σ map is simply constructed from the `enum` declarations in the program.
[5] The calculus does not have tuples, but the extension to tuples and polymorphic enums is straightforward. They are supported in the implementation.

$\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\, e : \tau_1 \to \tau_2} \text{ (T-Abs)}$$

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \equiv_{\mathbb{B}} \tau_2}{\Gamma \vdash e : \tau_2} \text{ (T-Eq)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2} \text{ (T-App)}$$

$$\frac{}{\Gamma \vdash () : \mathsf{Unit}} \text{ (T-Unit)}$$

$$\frac{}{\Gamma \vdash \mathtt{true} : \mathsf{Bool}} \text{ (T-True)}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \mathsf{gen}(\Gamma, \tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \mathsf{let}\, x = e_1\, \mathsf{in}\, e_2 : \tau_2} \text{ (T-Let)}$$

$$\frac{}{\Gamma \vdash \mathtt{false} : \mathsf{Bool}} \text{ (T-False)}$$

$$\frac{\Gamma \vdash e : \mathcal{E}[\varphi]}{\Gamma \vdash \mathsf{open}\, e : \mathcal{E}[\varphi \cup \varphi']} \text{ (T-Open)}$$

$$\frac{(x, \sigma) \in \Gamma \quad \sigma \sqsubseteq \tau}{\Gamma \vdash x : \tau} \text{ (T-Var)}$$

$$\frac{\Gamma \vdash e : \tau \quad \Sigma(\mathcal{E}.\ell) \sqsubseteq \tau \to \mathcal{E}[\varphi \cup \{\mathcal{E}.\ell\}]}{\Gamma \vdash \mathcal{E}.\ell(e) : \mathcal{E}[\varphi \cup \{\mathcal{E}.\ell\}]} \text{ (T-Tag)}$$

$$\frac{\eta_i = \mathsf{case}\, \mathcal{E}.\ell_i(x_i) \Rightarrow e_i \quad \Gamma, x_i : \tau_i \vdash e_i : \tau_{\text{out}} \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.\ell_i\} \quad \Sigma(\mathcal{E}.\ell_i) \sqsubseteq \tau_i \to \mathcal{E}[\varphi_{\text{in}}]}{\Gamma \vdash \mathsf{choose}\, e\, \{\overline{\eta}\} : \tau_{\text{out}}} \text{ (T-Choose)}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathsf{if}\, e_1\, \mathsf{then}\, e_2\, \mathsf{else}\, e_3 : \tau} \text{ (T-Ite)}$$

$$\frac{\eta_i = \mathsf{case}\, \mathcal{E}.\ell_i(x_i) \Rightarrow e_i \quad \Gamma, x_i : \tau_i \vdash e_i : \mathcal{E}[\varphi_i^{\text{out}}] \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \Sigma(\mathcal{E}.\ell_i) \sqsubseteq \tau_i \to \mathcal{E}[\varphi_{\text{in}}] \quad \varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.\ell_i\} \quad \left(\varphi_{\text{in}} \cap \left(\bigcup_i \left(\varphi_i^{\text{out}} \cap \{\mathcal{E}.\ell_i\}\right)\right)\right) \cup \bigcup_i \left(\varphi_i^{\text{out}} - \{\mathcal{E}.\ell_i\}\right) <: \varphi_{\text{out}}}{\Gamma \vdash \mathsf{choose}\star\, e\, \{\overline{\eta}\} : \mathcal{E}[\varphi_{\text{out}}]} \text{ (T-Choose-}\star\text{)}$$

$$\mathsf{gen}(\Gamma, \tau) = \forall \overline{\alpha}.\tau \text{ where } \overline{\alpha} = \mathsf{ftv}(\tau) \setminus \mathsf{ftv}(\Gamma)$$

Figure 3 Type Rules for $\lambda^{\mathsf{res}}_{\mathsf{var}}$.

The (T-Open) rule allows tagged values to be typed with additional labels. Essentially, the (T-Open) rule enables a form of weakening, which is necessary for the proof of preservation, without having to introduce general sub-typing into the system, since that would break type inference (recall that the surface language does not have $\mathsf{open}\, e$ expressions).

The (T-Choose) rule states that a choose expression of the form $\mathsf{choose}\, e\, \{\overline{\eta}\}$, where $\eta_i = \mathsf{case}\, \mathcal{E}.\ell_i(x_i) \Rightarrow e_i$, can be typed as τ_{out}, if the scrutinee e has the type $\mathcal{E}[\varphi_{\text{in}}]$, each tag's type scheme can be instantiated to $\tau_i^{in} \to \mathcal{E}[\varphi_{\text{in}}]$, φ_{in} is less than the union of the handled tags $\mathcal{E}.\ell_i$, if each result e_i has the type τ_{out} under an environment where x_i has type τ_i. This rule expresses the standard match typing conditions, but allows non-exhaustive matches as long as the type of e ensures that the value of e will be handled. In this rule we see why the scheme of labels do not include their tag. If it was included, then φ_{in} would *have* to include the label of the case.

The (T-Choose-$\star$) is similar to the (T-Choose) rule but with two major differences: First, the type of each result e_i must be of the form $\mathcal{E}[\varphi_i^{\text{out}}]$, and second, a side-condition is posed relating the input and output. The side-condition requires the output φ_{out} to be greater than a union of two set formulas: The first formula represents the set of labels *maintained* in φ_{in}; i.e. the labels in the type of the cases that matches the label of the case and also exists in φ_{in}. The second formula represents the set of labels *introduced* by each case; i.e. the labels in the type of the cases that does not match the label of the case.

We explain the additional side-condition in the (T-Choose-⋆) rule with an example. Assume that we have the program below on the left and we assign the case expressions the types on the right:

```
choose* c {
    case Red    => Red
    case Green  => Blue
    case Blue   => Green
}
```

$$\varphi_1^{\text{out}} = \mathsf{Red}$$
$$\varphi_2^{\text{out}} = \mathsf{Blue}$$
$$\varphi_3^{\text{out}} = \mathsf{Green}$$

If we instantiate the additional side-constraint (T-Choose-⋆), we get:

$$(\varphi_{\text{in}} \cap (\mathsf{R} \cap \mathsf{R}) \cup (\mathsf{G} \cap \mathsf{B}) \cup (\mathsf{B} \cap \mathsf{G})) \cup (\mathsf{R} - \mathsf{R}) \cup (\mathsf{G} - \mathsf{B}) \cup (\mathsf{B} - \mathsf{G}) <: \varphi_{\text{out}} \tag{1}$$

where we have highlighted the two parts of the outer union. This simplifies to:

$$(\varphi_{\text{in}} \cap \mathsf{R}) \cup (\mathsf{G} \cup \mathsf{B}) <: \varphi_{\text{out}} \tag{2}$$

That is, the result may contain Blue and Green, but whether it contains Red is dependent on whether the input contains Red.

If, instead of $\varphi_2^{\text{out}} = \{\mathsf{Blue}\}$, we had assumed $\varphi_2^{\text{out}} = \{\mathsf{Blue}\} \cup \beta$ then we get:

$$(\varphi_{\text{in}} \cap (\mathsf{Red} \cup (\mathsf{Green} \cap \beta))) \cup (\mathsf{Green} \cup \mathsf{Blue}) \cup (\beta - \mathsf{Green}) <: \varphi_{\text{out}} \tag{3}$$

This is sensible because if we later learn that $\beta = \mathsf{Yellow}$ (i.e. the second case could return Blue or Yellow) then the above type reduces to:

$$(\varphi_{\text{in}} \cap \mathsf{Red}) \cup (\mathsf{Green} \cup \mathsf{Blue} \cup \mathsf{Yellow}) <: \varphi_{\text{out}} \tag{4}$$

3.3 Uninhabited Types

The type system of $\lambda_{\text{var}}^{\text{res}}$ admits programs that have uninhabited types. For example:

```
def f(c) =                              // c must have type Color[s]
    choose c { case Red   => ... }; //   where s <: {Red}
    choose c { case Green => ... }  //   where s <: {Green}
```

Here, the two pattern-matches give rise to the constraints $s <: \{\mathsf{Red}\}$ and $s <: \{\mathsf{Green}\}$. Thus, the type of the formal parameter c is Color[{}] which is uninhabited. However, this is not a problem; it simply means we cannot call f.

3.4 Meta Theory

The meta theory for the type system is fairly straightforward. We want to ensure that programs which use `choose` and `choose*` cannot get stuck. In other words, we want to prove the standard progress and preservation theorems.

We begin with the canonical forms lemma extended with typing inversion. The lemma shows that the index of a tagged value over-approximates its label:

▶ **Lemma 9** (Canonical-Tag). *If a value is typed with an enum type then the value must be a label of that enum and the enum index includes the label of the value.*

If $\vdash v : \mathcal{E}[\varphi]$ *then for some* ℓ, v', τ_1, *and* φ' *it holds that:*

1. $v = \mathcal{E}.\ell(v')$
2. $\vdash v' : \tau_1$

3. $\varphi \equiv_{\mathbb{B}} \varphi' \cup \{\mathcal{E}.\ell\}$
4. $\Sigma(\mathcal{E}.\ell) \sqsubseteq \tau_1 \to \mathcal{E}[\varphi' \cup \{\mathcal{E}.\ell\}]$

Another key lemma shows that the $\mathsf{open}\, e$ expression enables a form of subtyping (which is used to prove preservation of `choose*`):

▶ **Lemma 10** (Open-Tag). *If a value can be typed as an enum with some index then it can also be typed with a super set of that index.*
If $\vdash \mathcal{E}.\ell(v) : \mathcal{E}[\varphi]$ *then* $\vdash \mathcal{E}.\ell(v) : \mathcal{E}[\varphi \cup \varphi']$.

▶ **Theorem 11** (Progress). *For any closed, well-typed expression then either it is a value or it can evaluate to another expression.*
If $\vdash e : \tau$ *then* $e \in \mathit{Val}$ *or* $e \rightsquigarrow e'$.

▶ **Theorem 12** (Preservation). *If a closed well-typed expression can take a step then the new expression can also be typed with the original type.*
If $\vdash e : \tau$ *and* $e \rightsquigarrow e'$, *then* $\vdash e' : \tau$.

The proofs are available in the extended version of the paper.

3.5 Type Inference

We can support type inference for $\lambda^{\mathsf{res}}_{\mathsf{var}}$ with a suitable extension of Algorithm W [8, 28] with Boolean unification on set formulas [27]. We can use the type rules from the declarative type system of Figure 3 to systematically obtain a collection of type inference rules. The declarative system uses a typing judgment of the form $\Gamma \vdash e : \tau$, the type inference system extends this to $\Gamma \vdash e : \tau; S$ where S is a substitution. Here the type environment Γ and the expression e can be seen as the input to the type inference algorithm and τ and S as the output. We omit the actual inference rules, but they mostly concern a lot of administration around the careful use of substitutions and the composition of substitutions. As is standard, equalities in the declarative system become unification queries in the inference system.

We solve unification queries on types in the standard way, but when we reach two Boolean set formulas we use Boolean unification to solve the queries. Specifically, we rely on the Successive Variable Elimination (SVE) algorithm [27]. The most interesting aspect is how we translate set formula constraints, in the declarative type rules, into unification queries. This however – *by design* – turns out to be straightforward. Given the (T-Choose) type rule:

$$\frac{\eta_i = \mathsf{case}\, \mathcal{E}.\ell_i(x_i) \Rightarrow e_i \quad \Gamma \vdash e : \mathcal{E}[\varphi_{\text{in}}] \quad \boxed{\varphi_{\text{in}} <: \bigcup_i \{\mathcal{E}.\ell_i\}} \quad \Gamma, x_i : \tau_i \vdash e_i : \tau_{\text{out}} \quad \Sigma(\mathcal{E}.\ell_i) \sqsubseteq \tau_i \to \mathcal{E}[\varphi_{\text{in}}]}{\Gamma \vdash \mathsf{choose}\, e\, \{\overline{\eta}\} : \tau_{\text{out}}} \quad \text{(T-Choose)}$$

The interesting part is to translate what is shown in the gray box. Recall that this is the part of the constraint which ensures that the input is upper-bounded by the labels that occur in the pattern-match. We translate this constraint to the Boolean unification query:

$$\varphi_{\text{in}} \bigcap \left(\bigcup_i \{\mathcal{E}.\ell_i\} \right)^{\complement} \stackrel{?}{=} \emptyset$$

whose most-general unifier will capture exactly the above property. Similarly, we can translate the additional side-condition in (T-Choose-$\star$) as a unification problem on set formulas.

At the time of writing, the type inference machinery works (c.f. Section 4), but sometimes the substitutions computed by SVE can be very large. Large substitutions lead to large formulas which leads to slow inference. Fortunately, we have good reason to believe that the situation can be improved. We know from the case studies (c.f. Section 6) that most functions have small types (i.e. small formulas). Hence the challenge is to compute them. We think that this should be possible with a more sophisticated implementation of SVE that exploits Boolean technology, such as BDDs or ZDDs [1, 29].

3.6 Subtyping

The type system does not have explicit support for subtyping, but instead, like row-based systems, relies on parametric polymorphism [16, 37]. For example, the if-then-else expression:

$\mathsf{if}\,(\mathsf{true})\,\mathsf{then}\,\mathsf{Red}\,\mathsf{else}\,\mathsf{Blue}$

is typable because we can assign the types:

$$\Gamma \vdash \mathsf{Red} : \mathsf{Color}[\{\mathsf{Red}, \mathsf{Blue}\} \cup s] \qquad \text{and} \qquad \Gamma \vdash \mathsf{Blue} : \mathsf{Color}[\{\mathsf{Blue}, \mathsf{Red}\} \cup s]$$

for some type variable s. We could probably extend the type system with subtyping, but then we would likely lose principal type inference.

3.7 A Few Practical Aspects

We conclude with a discussion of a few practical issues.

- When should a programmer use `choose` or `choose*`? A programmer should use `choose` when he or she wants to partially pattern-match on a subset of labels, but the result can be of any type. On the other hand, a programmer should reach for `choose*` when he or she wants to partially pattern-match on a restrictable variant and the result is *the same restrictable variant*. In this case, `choose*` is preferable because it is structure-preserving; relating the "input" labels to the "output" labels.
- Would it be possible to have one "universal" type that holds all possible variants? Yes, in the limit one could define a single gigantic restrictable variant with all possible labels and then use that type everywhere in the program. In practice, this would probably be cumbersome and confusing. For example, it would seem pointless to merge the Color and Expr restrictable variants, even though one could conceptually do so.

4 Implementation

We have implemented the $\lambda^{\mathsf{res}}_{\mathsf{var}}$ calculus as an extension of the Flix programming language.

Flix is a functional, imperative, and logic programming language that supports algebraic data types, pattern matching, higher-order functions, parametric polymorphism, type classes, higher-kinded types, first-class Datalog constraints, channel and process-based concurrency, and has a polymorphic type and effect system [23, 21, 22, 24]. The Flix compiler project, including the standard library and tests, is approximately 230,000 lines of code.

Adding restrictable variants required approximately 2,000 lines of code. Most of the code was straightforward; the most complex components were the implementation of the type inference rules and Boolean unification on set formulas.

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

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

5 Expressiveness and Comparison to Other Systems

In this section, we compare the expressiveness of restrictable variants to other nominal and/or structural type systems. We focus on type systems that support complete type inference. We remind the reader that in Section 2 we used restrictable variants to express:

```
def simplify(e: Expr[s]): Expr[(s - Xor) + {Not, And, Or}]
def subst(m: Map[Int32, Bool], e: Expr[s]): Expr[(s - Var) + Cst]
def fasteval(e: Expr[s & {Cst, Not, And, Or}]): Bool
```

which allowed us to use function composition to define:

```
let fastrun = m -> simplify >> subst(m) >> fasteval
```

We now discuss our ability to express this in other systems with extensible variants.

5.1 Row Polymorphism à la Wand, Gaster and Jones, and Leijen

Row polymorphism is a classic solution to extensible records and variants [37]. A row polymorphic type system supports three primitive operations [11, 16] on variants which are *injection*, *embedding*, and *decomposition*:

$$\langle \ell = \_ \rangle : \forall \alpha, r.\, \alpha \to \langle \ell : \alpha \mid r \rangle \qquad \text{(injection)}$$

$$\langle \ell \mid \_ \rangle : \forall \alpha, r.\, \langle r \rangle \to \langle \ell : \alpha \mid r \rangle \qquad \text{(embedding)}$$

$$(\ell \in \_\ ?\ \_ : \_) : \forall \alpha, \beta, r.\, \langle l : \alpha \mid r \rangle \to (\alpha \to \beta) \to (\langle r \rangle \to \beta) \to \beta \qquad \text{(decomposition)}$$

The last operation is allows us to implement pattern matching. What is important is that each use of the ternary-like conditional $(\ell \in \_\ ?\ \_ : \_)$ peels off a variant. Note that if we fail to match on ℓ then we refine the type to $\langle r \rangle$ which we continue with in the else branch.

Leijen gives the example [16]:

```
showEvent e =
    (key in e) ? (c -> showChar(c)) :
        (e' -> (mouse in e') ? (p -> showPoint(p)) : error())
```

Here the idea is that showEvent pattern-matches on an extensible variant of the type:

$$\langle \mathsf{key} : \mathsf{KeyEvent} \mid \mathsf{mouse} : \mathsf{MouseEvent} \rangle$$

using the decomposition operator. Note that the program type-checks because both functions showChar and showPoint (and error) return a value of the same type, i.e. Unit.

In any case, the return type of the entire "pattern-match" is β, which means that the returned values must have the same type (modulo row-equivalence). Looking over our three functions, we see that[6]:

We can express the eval and fasteval functions which are given the types:

$$\mathsf{eval} : \langle \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} : \cdot \mid \mathsf{Xor} : \cdot \rangle \to \mathsf{Bool}$$

$$\mathsf{fasteval} : \langle \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} : \cdot \rangle \to \mathsf{Bool}$$

[6] For simplicity, we ignore the fact that the data type is recursive. We just focus on the labels themselves.

Here the rows are closed and the two functions accept any Boolean formula as long as it only has one of the listed variants. In particular, we cannot accidentally call eval or fasteval with an open Boolean formula that has the Var label.

We can also express the simplify and subst functions which are given the types:

$$\begin{aligned}\mathsf{simplify} &: \langle \mathsf{Var} : \cdot \mid \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} : \cdot \mid \mathsf{Xor} : \cdot \rangle \to \\ &\quad \langle \mathsf{Var} : \cdot \mid \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} \rangle \\ \mathsf{subst} &: \mathsf{Map}[\mathsf{Int32}, \mathsf{Bool}] \to \langle \mathsf{Var} : \cdot \mid \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} : \cdot \mid \mathsf{Xor} : \cdot \rangle \to \\ &\quad \langle \mathsf{Cst} : \cdot \mid \mathsf{Not} : \cdot \mid \mathsf{Or} : \cdot \mid \mathsf{And} \mid \mathsf{Xor} : \cdot \rangle\end{aligned}$$

Each function accepts a Boolean expression as input, with any labels, and returns a Boolean formula without the Xor and Var labels, respectively. However, their row types *cannot capture how the input is related to the output*. Hence, unlike with restrictable variants, when we compose the two functions we lose information about the output. In particular, the information that simplify has eliminated the Xor variant is lost. Hence we cannot call fasteval.

One might wonder if we could give the simplify function the type:

$$\mathsf{simplify} : \langle \mathsf{Xor} : \cdot \mid r \rangle \to \langle r \rangle$$

since that seems to capture what we want. However, this type judgment would be *unsound* since we could instantiate r to $\langle \mathsf{Cst} : \mathsf{Int32} \rangle$ and clearly the implementation of simplify is not exhaustive for that label. Moreover, if we fail to mention e.g. Var then we could also instantiate r to $\langle \mathsf{Var} : \mathsf{Banana} \rangle$ which is also not sound. Thus the only sound solution must mention *all* the variants that simplify is prepared to accept. Thus we lose the relationship between the input and output.

A challenge with extensible records and variants based on rows is the question of whether extensions adds a new field (or variant) or overrides an existing field (or variant). The literature has proposal several solutions to this problem:

Extensible Records and Variants with Qualified Types

Gaster and Jones present a type system for extensible records and variants extended with qualified types [11, 13]. The idea is that rows capture the structure of the record (or variant) while predicates are used to ensure that rows are not extended with labels that are already present. Thus the Gaster and Jones system ensures that a record (or variant) cannot be extended with a label it already has. For example, record extension is given the type:

$$(l := \_ \mid \_) : (r \setminus l) \Rightarrow \alpha \to \mathsf{Rec}\, r \to \mathsf{Rec}\, \{l : \alpha \mid r\}$$

where the predicate $(r \setminus l)$ in the qualified type captures that r must not contain the label l.

Extensible Records with Scoped Labels

Leijen proposes a different approach that embraces the idea of duplicate labels in records and variants [16]. In Leijen's type system, extensible records are allowed to have multiple fields with the same name (and of different type). This means that fields are scoped.

For example, we can have a record r with the type:

$$r : \{y : \mathsf{Bool} \mid x : \mathsf{Int32} \mid y : \mathsf{Int32}\}$$

This means that r has *two* fields named y; one of type Bool and the other of type Int32. We can access the former, the outermost, with the expression $r.y$. To access the latter, the innermost, we must first remove the outermost y field. Thus we have to write $(r - y).y$. The advantage of Leijen's over Gaster and Jones's is that it has principal types without the need for qualified types. The disadvantage is its somewhat unnatural semantics.

What does this mean for a variant to be scoped? It roughly means that when we pattern-match (i.e. decompose) on an extensible variant we always see the outermost label. To find an inner label, we must decompose once, and then decompose again. In pseudo-code:

```
match v /* has type: { Label: Int32 | Label: String | ... } */ {
    case Label(n) => n + 123
    case rest => match rest /* has type: { Label: String | ... } */ {
        case Label(s) => String.toUpperCase(s)
    }
}
```

where we peel off one layer of v to expose the inner Label of type String. As Leijen writes, this is a "curious" feature and it is not so clear whether it is useful in practice.

Abstracting Extensible Data Types à la Morris et al.

Morris and McKinna presents a type system that unifies the previous type systems into one framework based on qualified types and row theories [30]. The framework can be instantiated to model the systems of Gaster and Jones and Leijen among others. Importantly, the framework also supports instantiations with row concatenation.

The framework can also support a form of extensible variants with the key constructs:

$$
\begin{array}{rcll}
\lambda\, x\, .\, \ell \triangleright x & : & \tau \to \Sigma(\ell \triangleright \tau) & (\textsc{Construction}) \\
\lambda\, x\, .\, x/\ell & : & \Sigma(\ell \triangleright \tau) \to \tau & (\textsc{Extraction}) \\
\lambda\, x\, .\, \texttt{inj}\ x & : & \forall\ z_1\ z_2\ .\ z_1 \lessdot z_2 \Rightarrow \Sigma z_1 \to \Sigma z_2 & (\textsc{Injection}) \\
\lambda\, x\, y\, .\, x \bigtriangledown y & : & \forall\ z_1\ z_2\ z_3\ \tau\ .\ z_1 \circledcirc z_2 \sim z_3 \Rightarrow & \\
 & & \quad (\Sigma z_1 \to \tau) \to (\Sigma z_2 \to \tau) \to (\Sigma z_3 \to \tau) & (\textsc{Match})
\end{array}
$$

The key idea is the use of qualified types with two predicates: the *containment* predicate $z_1 \lessdot z_2$ and the *combination* predicate $z_1 \circledcirc z_2 \sim z_3$. Using these qualified type predicates, we can define the operations:

- (Construction) constructs a singleton variant with the label ℓ and type τ.
- (Extraction) destructs a *singleton* variant and extracts the value of the variant.
- (Injection) extends a variant with additional labels. The operation uses the *containment* predicate $z_1 \lessdot z_2$. Its meaning is dependent on the specific row theory. For example, using a theory that disallows duplicates, it means that the labels of z_1 must be a subset of the labels of z_2 (with compatible types).
- (Match) is a combinator that composes two functions which operate on parts of a variant into a single function that works on the row concatenation of their input types. The operation uses the *combination* predicate $z_1 \circledcirc z_2 \sim z_3$. Its meaning is again dependent on the specific row theory. For example, using a theory that disallows duplicates, it means that z_1 and z_2 must be disjoint sets of labels and z_3 must be their union. Note that the return type τ of the two functions must be same, hence the match construct does not relate its input type to its output type.

We illustrate this loss of precision with the following example:

$$\lambda\ \mathtt{x}\ .\ \left(\left(\lambda\ \mathtt{y}\ .\ \mathtt{inj}\ (A \triangleright \mathtt{y}/A)\right) \triangledown \left(\lambda\ \mathtt{z}\ .\ \mathtt{inj}\ (B \triangleright \mathtt{z}/B)\right)\right)\left(\mathtt{inj}\ \mathtt{x}\right) \quad :$$

$$\forall z_1, z_2, \tau_1, \tau_2\ .\ z_1 \lessdot (A \triangleright \tau_1, B \triangleright \tau_2), (A \triangleright \tau_1, B \triangleright \tau_2) \lessdot z_2 \Rightarrow \Sigma z_1 \to \Sigma z_2$$

Informally, the function is simple the identity function on a variant with two labels A and B, i.e. it maps A to A and B to B. Assume – without loss of generality – that we work on a row theory based on Gaster and Jones which does not allow duplicate labels in variants.

The function receives a variant $\mathtt{x}$, which is allowed to be a subset of the variant $\Sigma(A \triangleright \tau_1, B \triangleright \tau_2)$. It is first injected to be typable with the complete variant, then it is matched on in two different functions that either assume that the variant was A or B via their respective extraction (y/A or z/B). Lastly, the variant is constructed again in singleton variants and injected into the full variant type.

The intention of the function is clearly shown in the type; the input must be a subset of a variant with A and B and the output must be a superset of a variant with A and B. While this type is correct, it is unfortunately not as a precise as we would have hoped. In particular, since the function is actually the identity we would have liked the type: $\forall z.\, z \lessdot (A \triangleright \tau_1, B \triangleright \tau_2) \Rightarrow \Sigma z \to \Sigma z$.

The type systems of Gaster and Jones, Leijen, and Morris and McKinna do not solve the fundamental "co-domain problem" for extensible variants. Rather they expose difficulties with row-based variants which require additional machinery or unnatural semantics to fix. Restrictable variants do not suffer from such issues because they rely on *set formulas*.

5.2 Occurrence Typing à la Castagna

Castagna et al. present an expressive set-theoretic type system with a type-case expression [7]. The type system supports union, intersection, and negation types. In their system, the Color type can be represented as the *union* type of three singleton types:

$$\mathsf{Red} \vee \mathsf{Green} \vee \mathsf{Blue}$$

and we can match on these using the type-case expression:

$$e_1 \in \tau\ ?\ e_2 : e_3$$

where control flow enters the e_2 branch if e_1 reduces to a value $v : \tau$, or e_3 if it does not; i.e. $v : \neg\tau$. The type-case expression is their powerful alternative to the *if-then-else*/*match*/*choose* expression, allowing an association between each possible type of the input and the respective type of the output. For example, in their system, the isWarm function can be expressed as:

$$\lambda x.\ x \in \mathsf{Red}\ ?\ \mathsf{True} : (x \in \mathsf{Blue}\ ?\ \mathsf{False} : \mathsf{undefined})$$

which has the type:

$$(\mathsf{Red} \to \mathsf{True}) \wedge (\neg\mathsf{Blue} \to \mathsf{False})$$

Note that, in the last case, where $x \notin \mathsf{Red}$ and $x \notin \mathsf{Blue}$ the untypable expression undefined is used to indicate an unreachable case. The precision of the typing – essentially encoding the entire pattern-match at the type level – is very expressive and solves the "co-domain problem" we have outlined. However, the types can become very complex and unwieldy, and there is limited support recursive types and recursive functions [7]. For example, the $\mathsf{subst}(m)$ function would be given the large intersection type:

$(\mathsf{Var}(\mathsf{Int32}) \vee \mathsf{Cst}(\mathsf{Bool}) \to \mathsf{Cst}(\mathsf{Bool})) \wedge (\mathsf{Not}(\mathsf{Expr}) \to \mathsf{Not}(\mathsf{ClosedExpr})) \wedge (\mathsf{Or}(\mathsf{Expr}, \mathsf{Expr}) \to \mathsf{Or}(\mathsf{ClosedExpr}, \mathsf{ClosedExpr})) \wedge (\mathsf{And}(\mathsf{Expr}, \mathsf{Expr}) \to \mathsf{And}(\mathsf{ClosedExpr}, \mathsf{ClosedExpr})) \wedge (\mathsf{Xor}(\mathsf{Expr}, \mathsf{Expr}) \to \mathsf{Xor}(\mathsf{ClosedExpr}, \mathsf{ClosedExpr}))$

where we also have to define the Expr and ClosedExpr data types as two large union types.

While the goal of $\lambda^{\mathsf{res}}_{\mathsf{var}}$ is to capture the introduction and elimination of variants, the occurrence typing system goes far beyond this, capturing a large amount of additional information as it maps variant to variant; the cost of the additional information is borne in the complexity of the types. Furthermore, it is not clear that the occurrence typing system is capable of inferring the type of recursive functions, meaning that in order to capture the same *elimination* and *introduction* properties, the programmer would have to provide the large type annotations themselves.

5.3 Relational Nullable Types à la Madsen et al.

Madsen and van de Pol present a relational nullable type system [25]. The type system captures the nullability (i.e. whether an expression may evaluate to null) of an expression in relation to the nullability of other related expressions. For example, using their type system, one can express a function:

```
let f = (host, port) -> match (host, port) {
    case (Absent, Absent)          => ...
    case (Present(h), Present(p)) => ...
}
```

which captures that *either* both host and port are Absent (i.e., "null") *or* both host and port are Present (i.e., non-"null"). For example, the following two calls type-check:

```
f(Absent, Absent)                          // OK
f(Present("www.google.com"), Present(80)) // OK
```

whereas the next two calls are rejected by the type system:

```
f(Absent, Present(80))                     // NOT OK
f(Present("www.google.com"), Absent)       // NOT OK
```

The relational nullable type system associates every expression with a proper type π and a pair of Boolean formulas (φ, ψ) that over-approximate whether the expression *may* evaluate to Absent (i.e., null) and *may* evaluate to Present (i.e., non-null) [25]. The two Boolean formulas form a small lattice where: String ? (F, F) is an uninhabited type (i.e., a type that is neither null nor non-null), and e.g. String ? (F, ψ) is the type of non-null Strings.

Relational nullable types and restrictable variants share some similarities:

- The restrictable variants type system use *one* type-level *set formula* to over-approximate the set of variants of an expression, whereas the relational nullable type system uses *two* type-level *Boolean formulas* to over-approximate the nullability and non-nullability of an expression.
- Both systems extend Hindley-Milner with Boolean unification; their system on Boolean formulas and our system on set formulas.
- We find that the relational nullable types tend to be significantly more complex than restrictable variant types. For example, the function from above is given the type:

$$\forall t_1, t_2, t_3, b_1, b_2, b_3, b_4.\, (t_1, b_1 \wedge \neg b_3 \wedge \neg b_4, b_3) \to (t_2, b_2 \wedge \neg b_3 \wedge \neg b_4, b_4) \to t_3$$

5.4 Summary

We believe that restrictable variants offer a new simple and practical sweet-spot in the design space of "extensible" data types. In terms of expressive power, for the programming patterns we have shown, we identify restrictable variants as laying between row-based type systems and full-blown occurrence typing. Importantly, restrictable variants precisely capture the introduction and elimination of variants which leads to better compositionality than row-based variants.

6 Case Studies

We now report on three small case studies that use restrictable variants. The first is the running example of Boolean formulas. The second is a new data structure that combines the Option, List, and NonEmptyList data types. The third is a theoretical study of how restrictable variants can be combined with extensible records to model abstract syntax trees.

6.1 Case Study: Boolean Expressions

We have seen how we can use restrictable variants to represent Boolean formulas. The key idea is that we can use the same data type represent both simple formulas (made from the Not, And, Or connectives) and more complex formulas (e.g. using the Xor connective). We can also represent both open and closed formulas (i.e. formulas with or without Vars).

6.2 Case Study: Option, List, and NonEmptyList

The Flix standard library supports the three central functional data types: Options, Lists, and Nels (non-empty lists). The Option module offers 75 functions and spans 587 lines of code, the List module offers 136 functions and spans 1,398 lines of code, and finally the Nel module offers 104 functions and spans 703 lines of code. While this "batteries included" approach is great for programmers, the downside is that the implementations of Option, List, and Nel duplicate a lot of functionality. Given that Option, List, and Nel are really just sequences of different lengths ($0-1$ for Option, $0-n$ for List, and $1-n$ Nel), one might wonder if they could not be unified into one data type. As it turns out, they can!

We can define *one data type* for sequences of integers[7]:

```
enum Seq[s] {
    case Nil
    case One(Int32)
    case Cons(Int32, Seq[s])
}
```

We can then define Option, List, and Nel as type aliases:

```
type alias Option = Seq[{Nil, One}]
type alias List   = Seq[{Nil, Cons}]
type alias Nel    = Seq[{One, Cons}]
```

[7] Flix naturally supports polymorphic data types, but for simplicity we focus on integer-valued sequences.

A slightly more general type would be e.g., `type alias Option[s] = Seq[s & {Nil, One}]`. What's important is that we can define common operations on Seq *once* and reuse them for different types of sequences.

For example, we can write a forall function:

```
def forall(f: Int32 -> Bool, s: Seq[s]): Bool = choose s { ... }
```

And we can also can write a map function:

```
def map(f: Int32 -> Int32, s: Seq[s]): Seq[s] = choose* s { ... }
```

Importantly, the map function preserves information about what variants can occur in the output based on the input. Thus, if we map over an Option, we know that the result is an Option and if we map over a Nel, we know the result is a Nel.

We can also write functions that only work for non-empty lists. For example:

```
def head(s: Seq[s - Nil]): Int32 = choose s { ... }
def last(s: Seq[s - Nil]): Int32 = choose s { ... }
```

More interestingly, we can express a function that appends an element to a sequence:

```
def append(elm: Int32, s: Seq[s]): Seq[{One, Cons}] = choose* s {
    case Nil          => One(w)
    case One(x)       => Cons(x, One(elm))
    case Cons(x, xs) => Cons(x, append(elm, xs))
}
```

The return type of append, which is equivalent to Nel, captures that the result lacks the Nil variant, hence is non-empty. We can use append to write a reverse function:

```
def reverse(s: Seq[s]): Seq[(s & {Nil}) + {One, Cons}] = choose* s {
    case Nil          => Nil
    case One(x)       => One(x)
    case Cons(x, xs) => append(x, reverse(xs))
}
```

The type of the reverse function is not as precise as we would like. In particular, if we reverse an Option type, we lose the information that the sequence has $0 - 1$ elements. However, the type is sufficiently precise to capture that if we reverse a non-empty list then the result is also non-empty.

In summary, in our experience, most aggregation functions such as head, forall, and count can be implemented on the Seq data type. We can also implement structure preserving functions such as map. Where it gets more difficult is with transformations such as append, reverse, and flatMap which do not always have the types we would want. In such cases, we can sometimes implement $2 - 3$ functions (corresponding to one for Option, List, and Nel) and thus still have the desired functionality.

6.3 Case Study: Restrictable Variants, Extensible Records

While we have presented *restrictable variants* as a better alternative to *extensible variants*, we have found that it is natural to combine restrictable variants with *extensible records*. For example, returning to the compiler use case, one can imagine an abstract syntax tree that is transformed and decorated with additional information through several compiler phases. We can use restrictable variants to capture the active labels and extensible records to capture the extra information. For example, we can define an abstract syntax tree:

```
enum Expr[s][r: RecordRow] {
    case Cst({value = Bool | r}),
    case Num({value = Int32 | r}),
    case Var({ident = String | r}),
    case Add({e1 = Expr[s, r], e2 = Expr[s, r] | r}),
    case Ite({e1 = Expr[s, r], e2 = Expr[s, r], e3 = Expr[s, r] | r})
    // ...
}
```

Here the the Expr data type has *two* type-level indices: The s index controls the variant part whereas the r index controls the record part. Assume that we also have a data type:

```
enum Type { case TBool, case TInt }
```

then we can use row extension to capture that type inference decorates the AST:

```
def infer(e: Expr[s][r]): Expr[s, (tpe = Type | r)] = ...
```

At the same time, we can also capture that code generation only works for closure-converted, lambda-lifted, and well-typed ASTs:

```
def codeGen(e: Expr[s - Lam][(tpe = Type | r)]): ByteCode = ...
```

This example illustrates that restrictable variants and extensible records complement each other well. We use the variant index to constrain what cases we are prepared to deal with and we use the record index to constrain what additional information we need.

6.4 Pretty Printing Types with Lower– and Upper Bounds

Programmers might find it difficult to read a type signature like:

```
def reverse(s: Seq[s]): Seq[(s & {Nil}) + {One, Cons}]
```

For this reason, we have experimented with showing lower- and upper-bounds of type-level set formulas. For example, the set formula: `Seq[(s & {Nil}) + {One, Cons}]` has the lower-bound: `{One , Cons}` and the upper-bound: `{Nil, One , Cons}`. This means a `choose` or `choose*` must handle the One and Cons variants and may optionally handle the Nil variant.

7 Related Work

We have already discussed how the expressiveness of restrictable variants compares to several other existing systems. In this section, we aim to provide high-level overview of related work.

Row-based Extensible Records and Variants

Wand originally introduced the concept of row variables for an object-oriented setting [37]. A key challenge in the literature on row-based systems has been how to deal with duplicate labels. A challenge that remains to this day [30]. Gaster and Jones present a type system for extensible records and variants that use qualified types [13] to ensure that rows do not contain duplicate labels [11]. Leijen instead propose a type system that permits duplicate labels and gives a semantics to such "scoped" records and variants [16]. Another major challenge has been the question of *row concatenation* [30]. In this direction, Harper and Pierce presents a record calculus and type system that permits record concatenation [12], but lacks type inference. Morris and McKinna presents a unified framework for row-polymorphic type systems based on *row theories* [30].

Row-based type systems have been used successfully in many applications other than extensible records and variants. For example, type systems based on rows have been used to track exceptions [31], to track effects in algebraic effect systems [17, 18], to model database queries [19], and to type first-class Datalog program values [22].

We refer to Gaster and Jones for a detailed introduction to row polymorphism [11].

Occurrence Typing, GADTs, Constructor Subtyping, and Relational Nullability

Castagna et al. present an occurrence-based type system [7] which uses set-theoretic types to infer precise function signatures. Applied to variants and pattern-matching, the system can track exactly how a function maps labels among each other. The system is purely structural and based on semantic subtyping, whereas our system includes nominal typing.

Generalized algebraic data types (GADTs) extend algebraic data types with additional expressive power by allowing the type scheme of a constructor to restrict its return type [15, 35]. The canonical example is the ability to write an algebraic data type for arithmetic and Boolean expressions $\mathsf{Expr}[\alpha]$ and an evaluation function $\mathsf{eval} : \mathsf{Expr}[\alpha] \to \alpha$ where α is a type-level index that determines whether the expression evaluates to a `Bool` or `Int`. A significant body of work has focused on how to recover type inference in the presence of GADTs [14, 32]. We think it would be interesting future work to explore possible connections between restrictable variants and GADTs.

Constructor subtyping is an alternative to extensible and restrictable variants where one inductive type τ_1 is considered a subtype of another inductive type τ_2 if τ_2 has more constructors than τ_1 [3, 26]. In relation to restrictable variants, the idea would be to have multiple data types that share similar constructors and then use subtyping to allow functions to operate on multiple of these types.

Madsen and van de Pol propose a type system with support for *relational nullable types* [25]. While a nullable type system tracks whether an expression may evaluate to null based on its type, *relational* nullable type systems track whether an expression may evaluate to null based on its type and the type of *other* related expressions. As discussed, the Madsen and van de Pol system has some similarities to ours: both systems allow partial (non-exhaustive) pattern-matching and both systems are based on Hindley-Milner extended with Boolean unification. However, their system is purely structural and focuses on nullability, whereas our system combines nominal and structural typing.

Refinement Kinds

Luís and Toninho propose refinement typing at the kind level to enable metaprogramming with records [6]. We believe their system could be adapted to variants and pattern matching: The dependent types in their system precisely track the associations between the input and output types of functions. Refinement kinds, however, do not support type inference.

Boolean Unification

Boole studied Boolean unification in the single-variable case and presented a simplified version of the successive variable elimination algorithm [4]. Later, Löwenheim presented another Boolean unification algorithm [20]. Today, an accessible introduction to Boolean unification is provided by Martin and Nipkow [27]. Additional background information is provided by Baader [2], Boudet et al. [5], Robinson and Voronkov [34].

Boolean unification was first used in a type system by de Vries et al. who used it model uniqueness [9]. Later, Madsen and van de Pol presented a polymorphic type and effect system which used Boolean unification for inference [24].

8 Conclusion

We have presented *restrictable variants* as a simple and practical alternative to extensible variants. A restrictable variant is a sum type indexed by a type-level set formula of its active labels. We have also introduced the `choose` and `choose*` pattern-matching constructs which enable non-exhaustive patterns matches on restrictable variants. Notably, the `choose*` construct allow us to precisely track the *introduction* and *elimination* of variants through function composition.

We have presented a type system for a minimal calculus with restrictable variants. The type system, which based on Hindley-Milner extended with type-level set formulas, ensures that non-exhaustive pattern-matches cannot get stuck. The system supports complete inference via a suitable extension of Algorithm W with Boolean unification on set formulas.

We have implemented restrictable variants as an extension of the Flix programming language and used the implementation for a few case studies. The extension is ready for use, freely available, and open-source.

References

1 Sheldon B. Akers. Binary decision diagrams. *IEEE Transactions on computers*, C-27(06), 1978. doi:10.1109/TC.1978.1675141.

2 Franz Baader. On the complexity of Boolean unification. *Information Processing Letters*, 67(4), 1998. doi:10.1016/S0020-0190(98)00106-9.

3 Gilles Barthe and Maria João Frade. Constructor subtyping. In *Programming Languages and Systems: 8th European Symposium on Programming (ESOP)*, 1999. doi:10.1007/3-540-49099-X_8.

4 George Boole. *The mathematical analysis of logic*. Macmillan, Barclay and Macmillan, 1847.

5 Alexandre Boudet, Jean-Pierre Jouannaud, and Manfred Schmidt-Schauss. Unification in Boolean rings and abelian groups. *Journal of Symbolic Computation*, 8(5), 1989. doi:10.1016/S0747-7171(89)80054-9.

6 Luís Caires and Bernardo Toninho. Refinement kinds: Type-safe programming with practical type-level computation. *Proc. of the ACM on Programming Languages*, 3(OOPSLA), 2019. doi:10.1145/3360557.

7 Giuseppe Castagna, Mickaël Laurent, Kim Nguyen, and Matthew Lutze. On type-cases, union elimination, and occurrence typing. *Proc. of the ACM on Programming Languages*, 6(POPL), 2022. doi:10.1145/3462306.

8 Luis Damas. *Type assignment in programming languages*. PhD thesis, The University of Edinburgh, 1984.

9 Edsko de Vries, Rinus Plasmeijer, and David M Abrahamson. Uniqueness typing simplified. In *Implementation and Application of Functional Languages: 19th International Workshop (IFL)*, 2008. doi:10.1007/978-3-540-85373-2_12.

10 Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proc. of the ACM SIGPLAN 1991 conference on Programming language design and implementation (PLDI)*, 1991. doi:10.1145/113445.113468.

11 Benedict R Gaster and Mark P Jones. A polymorphic type system for extensible records and variants. Technical report, University of Nottingham, 1996.

12 Robert Harper and Benjamin Pierce. A record calculus based on symmetric concatenation. In *Proc. of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, 1991. doi:10.1145/99583.99603.

13 Mark P Jones. A theory of qualified types. *Science of Computer Programming*, 22(3), 1994. doi:10.1016/0167-6423(94)00005-0.

14 Simon Peyton Jones, Geoffrey Washburn, and Stephanie Weirich. Wobbly types: type inference for generalised algebraic data types. Technical report, University of Pennsylvania, 2004.

15 Andrew Kennedy and Claudio V. Russo. Generalized algebraic data types and object-oriented programming. In *Proc. of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005. doi:10.1145/1094811.1094814.

16 Daan Leijen. Extensible records with scoped labels. In *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming (TFP)*, 2005.

17 Daan Leijen. Koka: Programming with row polymorphic effect types. In *Proc. 5th Workshop on Mathematically Structured Functional Programming (MSFP)*, 2014. doi:10.4204/EPTCS.153.8.

18 Daan Leijen. Type directed compilation of row-typed algebraic effects. In *Proc. of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, 2017. doi:10.1145/3009837.3009872.

19 Sam Lindley and James Cheney. Row-based effect types for database integration. In *Proc. of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI)*, 2012. doi:10.1145/2103786.2103798.

20 Leopold Löwenheim. Über das auflösungsproblem im logischen klassenkalkul. In *Sitzungsberichte der Berliner Mathematischen Gesellschaft 7*, 1908.

21 Magnus Madsen. The principles of the Flix programming language. In *Proc. of the 2022 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)*, 2022. doi:10.1145/3563835.3567661.

22 Magnus Madsen and Ondřej Lhoták. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428193.

23 Magnus Madsen, Jonathan Lindegaard Starup, and Ondřej Lhoták. Flix: A meta programming language for Datalog. In *Proc. of the 4th International Workshop on the Resurgence of Datalog in Academia and Industry (Datalog 2.0)*, 2022.

24 Magnus Madsen and Jaco van de Pol. Polymorphic types and effects with Boolean unification. *Proc. of the ACM on Programming Languages*, 4(OOPSLA), 2020. doi:10.1145/3428222.

25 Magnus Madsen and Jaco van de Pol. Relational nullable types with Boolean unification. *Proc. of the ACM on Programming Languages*, 5(OOPSLA), 2021. doi:10.1145/3485487.

26 Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In *Proc. of the 32nd Symposium on Implementation and Application of Functional Languages (IFL)*, 2020. doi:10.1145/3462172.3462194.

27 Urusula Martin and Tobias Nipkow. Boolean unification - the story so far. *Journal of Symbolic Computation*, 7(3), 1989. doi:10.1016/S0747-7171(89)80013-6.

28 Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978. doi:10.1016/0022-0000(78)90014-4.

29 Shin-ichi Minato. Zero-suppressed BDDs for set manipulation in combinatorial problems. In *Proc. of the 30th Design Automation Conference (DAC)*, 1993. doi:10.1145/157485.164890.

30 J Garrett Morris and James McKinna. Abstracting extensible data types: or, rows by any other name. *Proc. of the ACM on Programming Languages*, 3(POPL), 2019. doi:10.1145/3290325.

31 François Pessaux and Xavier Leroy. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(2), 2000. doi:10.1145/349214.349230.

32 François Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2006. doi:10.1145/1111037.1111058.

33 Didier Rémy. Type inference for records in a natural extension of ML. Technical report, University of Pennsylvania, 1990.

34 Alan JA Robinson and Andrei Voronkov. *Handbook of automated reasoning*, volume 1. Elsevier and MIT Press, 2001.

35 Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(1), 2007. doi:10.1145/1180475.1180476.

36 Niki Vazou, Eric L Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement types for Haskell. In *Proc. of the 19th ACM SIGPLAN international conference on Functional programming (ICFP)*, 2014. doi:10.1145/2628136.2628161.

37 Mitchell Wand. Type inference for simple objects. In *Proc. of the Fourth Annual Symposium on Logic in Computer Science*, 1987.