The Algebra of Patterns

David Binder
University of Kent, Canterbury, UK

Lean Ermantraut
Radboud University, Nijmegen, The Netherlands

Abstract

Pattern matching is a popular feature in functional, imperative and object-oriented programming languages. Language designers should therefore invest effort in a good design for pattern matching. Most languages choose a *first-match semantics* for pattern matching; that is, clauses are tried in the order in which they appear in the program until the first one matches. As a consequence, the order in which the clauses appear cannot be arbitrarily changed, which results in a less declarative programming model. The declarative alternative to this is an *order-independent semantics* for pattern matching, which is not implemented in most programming languages since it requires more verbose patterns. The reason for this verbosity is that the syntax of patterns is usually not expressive enough to express the *complement* of a pattern. In this paper, we show a principled way to make order-independent pattern matching practical. Our solution consists of two parts: First, we introduce a *boolean algebra of patterns* which can express the complement of a pattern. Second, we introduce *default clauses* to pattern matches. These default clauses capture the essential idea of a fallthrough case without sacrificing the property of order-independence.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Type theory

Keywords and phrases functional programming, pattern matching, algebraic data types, equational reasoning

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.2

Related Version *Extended Version*: `https://arxiv.org/abs/2504.18920` [2]

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.10`

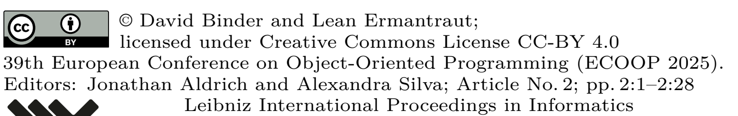
Funding *David Binder*: supported by ARIA programme for Safeguarded AI (grant MSAI-PR01-P11).

1 Introduction

Pattern matching is a programming language feature that combines two operations: Testing whether a value matches a given pattern and, if the test is successful, extracting subexpressions of the value and binding them to variables. A pattern matching expression consists of the *scrutinee* which is analyzed by the pattern match and a list of *clauses*. Each clause consists of a *pattern*, hence the name, and an expression called the *right-hand side*. These elements can all be seen in the following example of the length function, which computes the length of a list:

$$\text{length}(xs) \coloneqq \textbf{case}\ \underbrace{xs}_{\text{scrutinee}}\ \textbf{of}\ \{\ \underbrace{\texttt{Nil} \Rightarrow 0}_{\text{clause}},\ \underbrace{\texttt{Cons}(_, zs)}_{\text{pattern}} \Rightarrow \underbrace{1 + \text{length}(zs)}_{\text{right-hand side}}\ \}$$

Programming with pattern matching is popular. For a long time, only programmers who use functional languages could enjoy this feature, but today pattern matching has arrived in the mainstream of object-oriented and imperative programming languages. For example, Python, Java, Dart, and C# added pattern matching as an extension to the language, and some newer languages such as Rust and Swift, which are not primarily functional languages, supported pattern matching from the beginning.


© David Binder and Lean Ermantraut;
licensed under Creative Commons License CC-BY 4.0
39th European Conference on Object-Oriented Programming (ECOOP 2025).
Editors: Jonathan Aldrich and Alexandra Silva; Article No. 2; pp. 2:1–2:28
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Pattern matching has become one of the central constructs that is used to structure the control and data flow of a program. How pattern matches are checked and evaluated therefore fundamentally determines how we reason about the meaning of our programs; both informally when we read and write code in our daily lives, and formally in proof assistants, static analyzers and other verification tools. It is therefore important to choose a semantics for pattern matching which makes reasoning as simple as possible. One choice that language designers have to make is whether the clauses are tried in the order in which they appear in the program, or whether the order of the clauses does not matter. The following definition captures this difference:

▶ **Definition 1** (First-Match and Order-Independent Semantics). *A language uses* first-match semantics *if the clauses of a pattern match are tried in the order in which they occur until the first pattern matches the scrutinee. A language uses* order-independent semantics *if any of the matching clauses in a pattern match can be used to non-deterministically reduce a pattern matching expression.*

Overlapping clauses, i.e. two clauses with patterns that can match the same scrutinee, are a problem for order-independent semantics since a pattern-matching expression with overlapping clauses can non-deterministically reduce to two different results. The obvious solution is to enforce that two clauses must never overlap. But if we adopt this solution, then programmers usually have to write much more verbose patterns. For example, instead of the following function which checks whether the argument is the color `Red`

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ _ \Rightarrow \texttt{False}\ \},$$

the programmer has to write the following more verbose expression

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \texttt{Green} \Rightarrow \texttt{False},\ \texttt{Blue} \Rightarrow \texttt{False}\ \}.$$

The source of this verbosity is the poor expressiveness of patterns, which usually do not allow to express the *complement* of a pattern without exhaustively enumerating all the alternatives. In this example, we had to explicitly mention both `Blue` and `Green` to express the complement of the pattern `Red`. In languages that support or-patterns we can combine the last two clauses in the single clause $(\texttt{Green} \parallel \texttt{Blue}) \Rightarrow \texttt{False}$, but we still have to mention all constructors explicitly. It is unreasonable to ask programmers to always list all constructors of a data type, and they would rightfully reject a programming language which enforces this. Every proposal to use order-independent pattern matching in a realistic programming language therefore has to address this verbosity problem.

We propose two complementary solutions to the verbosity problem. In our system, the function can be written in the following ways:

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \neg\texttt{Red} \Rightarrow \texttt{False}\ \} \qquad \text{(Negation Pattern)}$$

$$\text{isRed}(c) \coloneqq \textbf{case}\ c\ \textbf{of}\ \{\ \texttt{Red} \Rightarrow \texttt{True},\ \textbf{default} \Rightarrow \texttt{False}\ \} \qquad \text{(Default Clause)}$$

To write the first variant we enrich the syntax of patterns with *negation patterns* which match whenever the negated pattern does not match. The two patterns are no longer overlapping, and at the same time, the pattern matching expression is not more verbose than our first version. Such negation patterns were already proposed under the name of "anti-patterns" [16, 17, 6] and are also mentioned by some other authors [19, 11]; the core contribution of this paper, however, is to consider them in the context of a complete *boolean algebra of patterns* which allows us to apply algebraic reasoning to patterns.

To write the second variant we enrich the syntax of pattern-matching expressions with *default clauses*[1]. A pattern matching expression can contain at most one such default clause (where **default** is a keyword instead of a pattern) which matches whenever none of the normal clauses matches the scrutinee. At first glance, the meaning of a default clause looks identical to the meaning of a clause with a wildcard pattern, but this is only the case for first-match semantics. If we use order-independent semantics, then we see that the meaning of a default clause does not correspond to a clause with a wildcard pattern, but to a clause that is headed by the negation of the disjunction of all other patterns in the pattern matching expression. In the compilation algorithm, however, default clauses are treated natively, so that we don't have to expand a default clause to a large pattern in the case of pattern matches with many clauses.

We are particularly interested in the kind of reasoning which is preserved when we add or remove clauses from a pattern match, or when we add or remove constructors from a data type. We will argue that this is a significant software engineering challenge that most pattern-based languages do not adequately address yet.

1.1 Algebraic Reasoning and Variable Binding

The main contribution of this work is to support a rich catalog of equivalences for patterns. In fact, we strive to treat patterns as a boolean algebra which consists of negation patterns, and-patterns, or-patterns, wildcard-patterns and absurd-patterns, featuring all the expected equivalences. This is challenging from the perspective of variable binding. For instance, we want the law of double negation, i.e. $\llbracket \neg\neg p \rrbracket \equiv \llbracket p \rrbracket$, to hold, but this entails that we cannot ignore variables that occur inside patterns in a negated context. We will demonstrate that we can achieve the desired equivalences by a variable binding definition that tracks whether variables occur under an even or odd number of negations. A related problem is to have well-defined variable binding in the context of or-patterns $p \mathbin{\|} p'$, which requires a careful definition of linearity to make sure that the same variables are bound in all cases of an or-pattern. We also describe the conditions under which we can replace a pattern $p_1 \mathbin{\|} p_2$ by the pattern $p_2 \mathbin{\|} p_1$; this transformation is not semantic preserving in most systems which implement or-patterns.

1.2 Overview and Contributions

The rest of this article is structured as follows:

- In Section 2 we motivate why the first-match semantics that most programming languages use for pattern matching fails to address several important reasoning and software engineering problems. We show how our contributions can improve upon this status quo.
- In Section 3 we present our first main contribution: the algebra of patterns. We describe the syntax of patterns, the rules for non-deterministically matching a pattern against a value, and the restrictions that characterize the subset of linear and deterministic patterns. We also prove the algebraic properties that characterize the semantic equivalence relation on patterns.

[1] Some programming languages support default clauses with similar semantics to what we propose. Ada, for example, enforces that clauses in a case statement are not overlapping, but allows an optional "other" clause with the same semantics as our default clause. The clauses that Ada allows are however much more restricted, since they only work for enums instead of algebraic data types.

- Section 4 introduces our second contribution, a small term language that contains pattern-matching expressions with default clauses. This section also introduces exemplary typing rules to the untyped patterns of Section 3 and the constructs of our term language; the compilation of patterns described in later sections, however, does not depend on patterns being typed.
- Not all extensions to pattern matching can be compiled to efficient code, but ours can. In Section 5 we show how to compile patterns by translating them to a variant of a disjunctive normal form. We use these normalized patterns in the compilation of pattern-matching expressions to decision trees, which we describe in Section 6.
- We discuss future work in Section 7, related work in Section 8 and conclude in Section 9.

The results in sections Section 3 have been formally verified in the proof assistant Rocq, and are made available as complementary material. Theorems and definitions which have been formalized are marked with the symbol 🐓.

2 Motivation

Most programming languages that implement pattern matching use a first-match semantics for which the order of clauses matters. Changing this state of affairs is our main motivation for introducing a more expressive set of algebraic patterns. Let us therefore explain why we think that first-match semantics is not a good default choice for declarative languages.

2.1 First-Match Semantics Weakens Equational Reasoning

Equational reasoning is one of the most important methods functional programmers use to reason about code. Equational reasoning allows us to show that two expressions are equivalent if we can obtain one from the other by a series of rewriting steps. Each rewriting step is justified by some equation between terms, and functions defined by pattern matching are one of the main sources of such equations. Take, for example, the definitions of the functions "id" and "map":

$$\mathrm{id}(x) \coloneqq x \tag{1}$$
$$\mathrm{map}(f, []) \coloneqq [] \tag{2}$$
$$\mathrm{map}(f, x :: xs) \coloneqq f(x) :: \mathrm{map}(f, xs) \tag{3}$$

We can show that mapping the identity function over a list returns the original list, i.e. that $\mathrm{map}(\mathrm{id}, xs) = xs$ holds, by using the individual clauses of the definition of "map":

$$\mathrm{map}(\mathrm{id}, []) =_{(2)} []$$
$$\mathrm{map}(\mathrm{id}, x :: xs) =_{(3)} \mathrm{id}(x) :: \mathrm{map}(\mathrm{id}, xs) =_{(1)} x :: \mathrm{map}(\mathrm{id}, xs) =_{(IH)} x :: xs$$

We have annotated each rewriting step with the equation that justifies it, and in the last step, we have used the induction hypothesis (IH). In this example, we have used the fact that every clause of the pattern match is a valid equation that we can use for rewriting. But this was only valid because the equations in the definition of "map" do not overlap. To see why this is essential, consider the example from the introduction again:

$$\mathrm{isRed}(\texttt{Red}) \coloneqq \texttt{True} \tag{4}$$
$$\mathrm{isRed}(_) \coloneqq \texttt{False} \tag{5}$$

If we are using these clauses as equations we can now show that True is equal to False, using the proof `True` $=_{(4)}$ isRed(`Red`) $=_{(5)}$ `False`. A human is of course very unlikely to make such a silly mistake, but if we cannot guarantee that every clause holds as a valid equality between expressions, then we also limit what automatic tools can do for us.

2.2 First-Match Semantics Complicates Reasoning About Change

Software engineering has been described as the problem of integrating programming over time [27]. This means that we not only have to understand the meaning of a program at a fixed point in time, but we also have to understand how the meaning of a program changes as it is developed and maintained. It should therefore be as simple as possible to reason about how the meaning of a program changes when a programmer adds a clause to a pattern match, removes a clause from a pattern match, adds a constructor to a data type, or removes a constructor from a data type. Let us see why first-match semantics complicates reasoning about these kinds of changes.

If we want to understand the consequences of adding or removing a clause from a pattern match, then we have to consider all clauses occurring both above and below the clause we are changing. Having to reason about the entire context of the clause makes it much harder to spot bugs in code reviews, since it is not uncommon that pattern matching expressions can span multiple pages of code. Order-independent semantics guarantees that it doesn't matter if we add, delete or modify a clause at the beginning, middle or end of a list of clauses.

The other kind of change that frequently occurs is that we add or remove a constructor from an existing data type. Whether we have to adjust existing pattern matches depends on how those have been written. Even if we enforce that clauses in a pattern matching expression must not overlap, we have multiple possibilities to write exhaustive pattern matches. Consider a data type "Group" which consists of administrators, registered users and guests. We want to guarantee that only administrators have write access, but we have two possibilities to write the function:

hasWriteAccess : Group $\to$ Bool	hasWriteAccess : Group $\to$ Bool
hasWriteAccess(`Admin`) $:=$ `True`	hasWriteAccess(`Admin`) $:=$ `True`
hasWriteAccess(`RegisteredUser` $\parallel$ `Guest`) $:=$ `False`	hasWriteAccess($\neg$`Admin`) $:=$ `False`

On the left side we have pattern matched exhaustively on the complement of `Admin`, whereas on the right side we have used a negation pattern. These two programs behave the same, but we can observe a difference when we add moderators as a fourth type of user. We have to revisit the function on the left, whereas the function on the right continues to compile.

Some programming languages have already started to add support for enforcing these kind of considerations. For example, Rust supports the `#[non_exhaustive]` attribute on type declarations[2]. This attribute restricts pattern matches and ensures that they continue to compile after a new constructor has been added to the type; annotating the Group type with such an attribute would disallow the definition on the left. Dually, OCaml warns against writing "fragile pattern matches". A fragile pattern can hide newly added constructors, which might have unintended consequences. The algebraic patterns presented in this paper provide the necessary tools to program in situations where the programmer wants to enforce either one of these dual restrictions. There is also some empirical evidence that programmers feel uneasy about using wildcard patterns in evolving codebases. For example, the authors

[2] Cp. doc.rust-lang.org/stable/reference/attributes/type_system.html.

of a recent study about the programming practices of functional programmers [21] write: “Participants felt similarly about the use of wildcards in pattern matching, which silently assign pre-existing behavior to new variants of an enumeration. P7 mentioned that, in their main codebase, wildcards are completely disallowed for this reason, even though they make programming more convenient.”

2.3 Compositional Pattern Matching and Extensible Data Types

Our main motivation in this paper is to obtain a more declarative programming model for standard nominal data types and ordinary pattern matching. The compilation algorithm that we are presenting, however, also works for polymorphic variants [10]. Polymorphic variants are used in the programming language OCaml, for example, to handle open sets of possible error conditions without having to declare a data type which lists all possible errors. Our algorithm does not presuppose a data type declaration with a fixed set of constructors which is necessary to check for exhaustiveness; we do not discuss exhaustiveness of patterns in the main part of this paper, but outline the necessary modifications to check for it in the extended version [2]. We think that the ideas and the algebraic patterns that we present in this paper are also useful for some proposed solutions of the expression problem. In the compositional programming approach [25, 29, 28], for example, we can create pattern matches whose clauses are distributed among multiple different modules. In that scenario it is especially important to guarantee that the contributing clauses which come from different modules do not overlap in their left-hand side.

3 The Algebra of Patterns

As our first contribution, we introduce an *algebra of patterns*. The syntax of patterns and values is specified in Figure 1. We assume that we have an infinite set CtorNames of constructor names such as `True`, `False` or `Nil`, and a set Variables which contains variables such as x, y or z. Each constructor $\mathcal{C}$ comes with an implicit *arity* n, and we sometimes write $\mathcal{C}^n$ to make this explicit. For example, the constructor `Cons` has arity 2 since it needs to be applied to two arguments. Patterns p consist of variable patterns x, constructor patterns $\mathcal{C}^n(p_1, \dots, p_n)$, and-patterns $p \,\&\, p$, or-patterns $p \parallel p$, wildcard-patterns _, absurd-patterns #, and negation patterns $\neg p$. Values consist of constructors applied to other values; the two-element list $\texttt{Cons}(\texttt{True}, \texttt{Cons}(\texttt{False}, \texttt{Nil}))$ is an example of a value. If a pattern p matches a value v we obtain a substitution σ. Such substitutions are formalized as lists of mappings $x \mapsto v$, and we write $\sigma_1 +\!\!+\, \sigma_2$ for the concatenation of two such lists. Not every list of mappings is a proper map since some lists contain multiple mappings for the same variable, but we will later prove that patterns from the subset of linear patterns produce substitutions that map a variable to at most one value. There are two functions that compute the free variables contained in a pattern. The function $\mathrm{FV}^e(p)$ computes the free variables that occur under an *even* number of negations, whereas $\mathrm{FV}^o(p)$ computes the variables under an *odd* number of negations.

Variable, constructor and wildcard patterns are standard, so we do not introduce them explicitly and turn directly to the more interesting absurd-patterns, or-patterns, negation-patterns, and and-patterns.

The absurd-pattern # is the dual of the wildcard-pattern _. Whereas the wildcard-pattern matches any value, the absurd-pattern never matches a value. We include the absurd-pattern in the syntax of patterns since we need a canonical bottom element of the boolean algebra. But there are also more practical reasons to include an absurd pattern. In

$$
\begin{array}{llll}
 & & \mathcal{C}, \mathcal{C}^n \in \textsc{CtorNames} \quad x, y, z \in \textsc{Variables} & \\
p & ::= & x \mid \mathcal{C}^n(p_1, \ldots, p_n) \mid p \,\&\, p \mid p \parallel p \mid _ \mid \# \mid \neg p & \textit{Patterns} \\
v & ::= & \mathcal{C}^n(v_1, \ldots, v_n) & \textit{Values} \\
m & ::= & x \mapsto v & \textit{Mapping} \\
\sigma & ::= & [] \mid m :: \sigma & \textit{Substitution}
\end{array}
$$

$$
\begin{array}{rlrl}
\mathrm{FV}^e(x) & := \{x\} & \mathrm{FV}^o(x) & := \emptyset \\
\mathrm{FV}^e(_) & := \emptyset & \mathrm{FV}^o(_) & := \emptyset \\
\mathrm{FV}^e(\#) & := \emptyset & \mathrm{FV}^o(\#) & := \emptyset \\
\mathrm{FV}^e(\neg p) & := \mathrm{FV}^o(p) & \mathrm{FV}^o(\neg p) & := \mathrm{FV}^e(p) \\
\mathrm{FV}^e(p_1 \parallel p_2) & := \mathrm{FV}^e(p_1) \cup \mathrm{FV}^e(p_2) & \mathrm{FV}^o(p_1 \parallel p_2) & := \mathrm{FV}^o(p_1) \cup \mathrm{FV}^o(p_2) \\
\mathrm{FV}^e(p_1 \,\&\, p_2) & := \mathrm{FV}^e(p_1) \cup \mathrm{FV}^e(p_2) & \mathrm{FV}^o(p_1 \,\&\, p_2) & := \mathrm{FV}^o(p_1) \cup \mathrm{FV}^o(p_2) \\
\mathrm{FV}^e(\mathcal{C}(p_1, \ldots, p_n)) & := \cup_{i=1}^n \mathrm{FV}^e(p_i) & \mathrm{FV}^o(\mathcal{C}(p_1, \ldots, p_n)) & := \cup_{i=1}^n \mathrm{FV}^o(p_i)
\end{array}
$$

Figure 1 The syntax of values, patterns and substitutions. We track whether variables occur under an even or odd number of negations.

the lazy programming language Haskell, for example, we sometimes have to write a pattern that never matches but forces the evaluation of an argument. This is often achieved by adding a guard clause that evaluates to false, but we can also use an absurd pattern for the same purpose.

An or-pattern $p_1 \parallel p_2$ matches a value if either p_1 or p_2 (or both) matches the value. Or-patterns introduce a complication for the compiler: we have to ensure that if values are bound to variables in p_1 during matching, then p_2 has to bind values of the same type to the same variables. The following example on the left shows how or-patterns without variables can be used to combine multiple clauses into one.

$$
\begin{array}{ll}
\textbf{data}\ \text{Day} \ = \texttt{Mo} \mid \texttt{Tu} \mid \texttt{We} \mid \texttt{Th} \mid \texttt{Fr} \mid \texttt{Sa} \mid \texttt{Su} & \\
\text{isWeekend} : \text{Day} \to \text{Bool} & \text{isWeekend} : \text{Day} \to \text{Bool} \\
\text{isWeekend}(\texttt{Sa} \parallel \texttt{Su}) := \texttt{True} & \text{isWeekend}(\texttt{Sa} \parallel \texttt{Su}) := \texttt{True} \\
\text{isWeekend}(\texttt{Mo} \parallel \texttt{Tu} \parallel \texttt{We} \parallel \texttt{Th} \parallel \texttt{Fr}) := \texttt{False} & \text{isWeekend}(\neg(\texttt{Sa} \parallel \texttt{Su})) := \texttt{False}
\end{array}
$$

We could also have written the second clause of the previous example using a wildcard pattern, but then the two patterns would overlap. We can get rid of the verbose second clause by using a negation pattern, which is illustrated on the right. A negation pattern matches if, and only if, the negated pattern does not match.

The last type of patterns we introduce are and-patterns $p_1 \,\&\, p_2$ which succeed if both p_1 and p_2 match the scrutinee. While or-patterns $p_1 \parallel p_2$ require that p_1 and p_2 bind the same set of variables, and-patterns $p_1 \& p_2$ require p_1 and p_2 to bind disjoint sets of variables. Many languages already have a special case of and-patterns: An @-pattern $x@p$ (some languages also use the keyword "as") is an and-pattern where the left side must be a variable. If we want to rewrite the previous example to return a string instead of a boolean value, then we could use the following and-patterns:

$$
\begin{array}{l}
\text{isWeekend} : \text{Day} \to \text{String} \\
\text{isWeekend}(x \,\&\, (\texttt{Sa} \parallel \texttt{Su})) := \text{show}(x) +\!\!+ \text{`` is on the weekend''} \\
\text{isWeekend}(x \,\&\, \neg(\texttt{Sa} \parallel \texttt{Su})) := \text{show}(x) +\!\!+ \text{`` is not on the weekend''}
\end{array}
$$

Here we use the show function to print a weekday and ++ for string concatenation. We will use this snippet as a running example in Section 5 and Section 6 to show how patterns are normalized and compiled to efficient code.

3.1 Matching Semantics of Patterns

We will now present the formal rules which specify what happens when we match a pattern against a value; these rules can be found in the upper half of Figure 2. We use two mutually recursive judgments to formalize the semantics of matching: The judgment $p \rhd v \rightsquigarrow \sigma$ expresses that the pattern p successfully matches the value v and also binds values to variables of the pattern; those mappings are recorded in the substitution σ. The judgment $p \ntriangleright v \rightsquigarrow \sigma$ expresses that the pattern p does *not* match the value v, and also produces a substitution. That the judgment $p \ntriangleright v \rightsquigarrow \sigma$ also produces a substitution might be surprising: How can there be a substitution if the value does not match the pattern? We can think of these as "temporarily deactivated" substitutions which will become available again if we enclose the pattern in an additional negation. In fact, such substitutions are necessary if we want to guarantee that the pattern p is semantically equivalent to $\neg\neg p$ in the case where p contains variables, for example in Equation (5) below.

▶ **Example 2.** These examples show some well-behaved instances of the matching judgments:

$$\mathrm{Cons}(x, xs) \rhd \mathrm{Cons}(2, \mathrm{Cons}(3, \mathrm{Nil})) \rightsquigarrow [x \mapsto 2, xs \mapsto \mathrm{Cons}(3, \mathrm{Nil})] \tag{1}$$

$$\mathrm{True} \parallel \mathrm{False} \rhd \mathrm{True} \rightsquigarrow [] \tag{2}$$

$$\mathrm{True} \ntriangleright \mathrm{False} \rightsquigarrow [] \tag{3}$$

$$\neg x \ntriangleright \mathrm{True} \rightsquigarrow [x \mapsto \mathrm{True}] \tag{4}$$

$$\neg\neg x \rhd \mathrm{True} \rightsquigarrow [x \mapsto \mathrm{True}] \tag{5}$$

Equation (1) illustrates how the rules Var and Ctor can be used to match a constructor pattern against a list while binding components of the list to variables x and xs. Equation (2) shows how the rule Or_1 (together with the rule Ctor) allows to successfully match the left side of an or-pattern against a value. Equation (3) illustrates the rule Ctor_2 and Equation (4) the combination of rules Var and Neg_2. Equation (5) shows why we need two mutually recursive judgments: in order for the pattern $\neg\neg x$ to match against the value True we need to use the rules Neg_1, Neg_2 and Var to ensure that the variable x is bound to the value True.

Using the rules for $p \rhd v \rightsquigarrow \sigma$ and $p \ntriangleright v \rightsquigarrow \sigma$ we can already prove that they are, in a certain sense, sound and complete:

▶ **Theorem 3** (Matching rules are sound, 🐓). *The rules from Figure 2 are sound. There is no pattern p, value v and substitutions σ_1, σ_2 such that both $p \rhd v \rightsquigarrow \sigma_1$ and $p \ntriangleright v \rightsquigarrow \sigma_2$ hold.*

▶ **Theorem 4** (Matching rules are complete, 🐓). *The rules from Figure 2 are complete, i.e. for any pattern p and value v there is some σ such that $p \rhd v \rightsquigarrow \sigma$ or $p \ntriangleright v \rightsquigarrow \sigma$ holds.*

Since we are interested in algebraic reasoning, we have to say when we consider two patterns to be semantically equivalent. We will define semantic equivalence in Definition 6 using the two judgments introduced above. But first, we have to define when two substitutions (which we formalized as lists of mappings) are semantically equivalent:

Pattern matches: $p \rhd v \rightsquigarrow \sigma$

$$\frac{}{x \rhd v \rightsquigarrow [x \mapsto v]}\ \textsc{Var} \qquad \frac{}{_ \rhd v \rightsquigarrow []}\ \textsc{Wild} \qquad \frac{p \not\rhd v \rightsquigarrow \sigma}{\neg p \rhd v \rightsquigarrow \sigma}\ \textsc{Neg}_1$$

$$\frac{p_1 \rhd v \rightsquigarrow \sigma}{p_1 \parallel p_2 \rhd v \rightsquigarrow \sigma}\ \textsc{Or}_1 \qquad \frac{p_2 \rhd v \rightsquigarrow \sigma}{p_1 \parallel p_2 \rhd v \rightsquigarrow \sigma}\ \textsc{Or}_2 \qquad \frac{p_1 \rhd v \rightsquigarrow \sigma_1 \qquad p_2 \rhd v \rightsquigarrow \sigma_2}{p_1 \,\&\, p_2 \rhd v \rightsquigarrow \sigma_1 \mathbin{+\!\!+} \sigma_2}\ \textsc{And}$$

$$\frac{p_1 \rhd v_1 \rightsquigarrow \sigma_1 \qquad \ldots \qquad p_n \rhd v_n \rightsquigarrow \sigma_n}{\mathcal{C}^n(p_1, \ldots, p_n) \rhd \mathcal{C}^n(v_1, \ldots, v_n) \rightsquigarrow \sigma_1 \mathbin{+\!\!+} \ldots \mathbin{+\!\!+} \sigma_n}\ \textsc{Ctor}$$

Pattern doesn't match: $p \not\rhd v \rightsquigarrow \sigma$

$$\frac{}{\# \not\rhd v \rightsquigarrow []}\ \textsc{Absurd} \qquad \frac{\exists i : p_i \not\rhd v_i \rightsquigarrow \sigma}{\mathcal{C}^n(p_1, \ldots, p_n) \not\rhd \mathcal{C}^n(v_1, \ldots, v_n) \rightsquigarrow \sigma}\ \textsc{Ctor}_1$$

$$\frac{p \rhd v \rightsquigarrow \sigma}{\neg p \not\rhd v \rightsquigarrow \sigma}\ \textsc{Neg}_2 \qquad \frac{\mathcal{C}^n \neq \mathcal{C}'^m}{\mathcal{C}^n(p_1, \ldots, p_n) \not\rhd \mathcal{C}'^m(v_1, \ldots, v_m) \rightsquigarrow []}\ \textsc{Ctor}_2$$

$$\frac{p_1 \not\rhd v \rightsquigarrow \sigma}{p_1 \,\&\, p_2 \not\rhd v \rightsquigarrow \sigma}\ \textsc{And}_1 \qquad \frac{p_2 \not\rhd v \rightsquigarrow \sigma}{p_1 \,\&\, p_2 \not\rhd v \rightsquigarrow \sigma}\ \textsc{And}_2 \qquad \frac{p_1 \not\rhd v \rightsquigarrow \sigma_1 \qquad p_2 \not\rhd v \rightsquigarrow \sigma_2}{p_1 \parallel p_2 \not\rhd v \rightsquigarrow \sigma_1 \mathbin{+\!\!+} \sigma_2}\ \textsc{Or}$$

Pattern is linear: $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$

$$\frac{}{x\ \mathbf{lin}^\pm}\ \textsc{L-Var}^\pm \qquad \frac{}{\#\ \mathbf{lin}^\pm}\ \textsc{L-Absurd}^\pm \qquad \frac{}{_\ \mathbf{lin}^\pm}\ \textsc{L-Wild}^\pm$$

$$\frac{p_{1,2}\ \mathbf{lin}^+ \qquad \mathrm{FV}^e(p_1) = \mathrm{FV}^e(p_2)}{p_1 \parallel p_2\ \mathbf{lin}^+}\ \textsc{L-Or}^+ \qquad \frac{p_{1,2}\ \mathbf{lin}^- \qquad \mathrm{FV}^o(p_1) \cap \mathrm{FV}^o(p_2) = \emptyset}{p_1 \parallel p_2\ \mathbf{lin}^-}\ \textsc{L-Or}^-$$

$$\frac{p_{1,2}\ \mathbf{lin}^+ \qquad \mathrm{FV}^e(p_1) \cap \mathrm{FV}^e(p_2) = \emptyset}{p_1 \,\&\, p_2\ \mathbf{lin}^+}\ \textsc{L-And}^+ \qquad \frac{p_{1,2}\ \mathbf{lin}^- \qquad \mathrm{FV}^o(p_1) = \mathrm{FV}^o(p_2)}{p_1 \,\&\, p_2\ \mathbf{lin}^-}\ \textsc{L-And}^-$$

$$\frac{p\ \mathbf{lin}^-}{\neg p\ \mathbf{lin}^+}\ \textsc{L-Neg}^+ \qquad \frac{p\ \mathbf{lin}^+}{\neg p\ \mathbf{lin}^-}\ \textsc{L-Neg}^-$$

$$\frac{p_i\ \mathbf{lin}^+ \qquad \bigcap_{i=1}^n \mathrm{FV}^e(p_i) = \emptyset}{C^n(p_1, \ldots, p_n)\ \mathbf{lin}^+}\ \textsc{L-Ctor}^+ \qquad \frac{p_i\ \mathbf{lin}^- \qquad \bigcup_{i=1}^n \mathrm{FV}^o(p_i) = \emptyset}{C^n(p_1, \ldots, p_n)\ \mathbf{lin}^-}\ \textsc{L-Ctor}^-$$

Pattern is deterministic: $p\ \mathbf{det}$

$$\frac{}{x\ \mathbf{det}}\ \textsc{D-Var} \qquad \frac{}{_\ \mathbf{det}}\ \textsc{D-Wild} \qquad \frac{p_1\ \mathbf{det}}{\neg p_1\ \mathbf{det}}\ \textsc{D-Neg} \qquad \frac{}{\#\ \mathbf{det}}\ \textsc{D-Absurd}$$

$$\frac{p_{1,2}\ \mathbf{det} \qquad p_1 \not\between p_2}{p_1 \parallel p_2\ \mathbf{det}}\ \textsc{D-Or}_1 \qquad \frac{p_{1,2}\ \mathbf{det} \qquad \mathrm{FV}^e(p_1) = \mathrm{FV}^e(p_2) = \emptyset}{p_1 \parallel p_2\ \mathbf{det}}\ \textsc{D-Or}_2$$

$$\frac{p_{1,2}\ \mathbf{det} \qquad \neg p_1 \not\between \neg p_2}{p_1 \,\&\, p_2\ \mathbf{det}}\ \textsc{D-And}_1 \qquad \frac{p_{1,2}\ \mathbf{det} \qquad \mathrm{FV}^o(p_1) = \mathrm{FV}^o(p_2) = \emptyset}{p_1 \,\&\, p_2\ \mathbf{det}}\ \textsc{D-And}_2$$

$$\frac{p_1\ \mathbf{det} \cdots\ p_n\ \mathbf{det}}{C^n(p_1, \ldots, p_n)\ \mathbf{det}}\ \textsc{D-Ctor}$$

Figure 2 Rules for matching patterns against values, and checking for linearity and determinism.

▶ **Definition 5** (Semantic equivalence of substitutions,). *Two substitutions σ and σ' are semantically equivalent if they contain the same mappings:*

$$[\![\sigma]\!] \equiv [\![\sigma']\!] \coloneqq \forall m, m \in \sigma \Leftrightarrow m \in \sigma'$$

We can now state the definition for semantic equivalence of patterns.

▶ **Definition 6** (Semantic equivalence of patterns,). *Two patterns p and q are semantically equivalent if they match the same values with equivalent substitutions, and if they also do not match the same values with equivalent substitutions:*

$$\begin{aligned} [\![p]\!] \equiv [\![q]\!] \coloneqq \forall v, \forall \sigma,\ & p \rhd v \leadsto \sigma \Rightarrow \exists \sigma', q \rhd v \leadsto \sigma' \wedge [\![\sigma]\!] \equiv [\![\sigma']\!] \\ \wedge\ & q \rhd v \leadsto \sigma \Rightarrow \exists \sigma', p \rhd v \leadsto \sigma' \wedge [\![\sigma]\!] \equiv [\![\sigma']\!] \\ \wedge\ & p \not\rhd v \leadsto \sigma \Rightarrow \exists \sigma', q \not\rhd v \leadsto \sigma' \wedge [\![\sigma]\!] \equiv [\![\sigma']\!] \\ \wedge\ & q \not\rhd v \leadsto \sigma \Rightarrow \exists \sigma', p \not\rhd v \leadsto \sigma' \wedge [\![\sigma]\!] \equiv [\![\sigma']\!] \end{aligned}$$

We have to include both judgment forms if we want semantic equivalence to be a congruence relation. To see why this is the case, suppose that we omit the second half of Definition 6. It would then follow that the two patterns $\#$ and $\neg x$ are semantically equivalent since they match against the same set of values (i.e. no values at all). But if we apply a negation-pattern to both patterns, then we can easily show that the resulting patterns are no longer semantically equivalent: $[\![\neg\#]\!] \equiv [\![_]\!] \not\equiv [\![x]\!] \equiv [\![\neg\neg x]\!]$. But since Definition 6 requires both patterns to agree on the values they match *and* don't match against, it actually defines a congruence relation on patterns:

▶ **Theorem 7** (Congruence,). *If $[\![p_1]\!] \equiv [\![p'_1]\!]$ to $[\![p_n]\!] \equiv [\![p'_n]\!]$ hold, then we also have $[\![\neg p_1]\!] \equiv [\![\neg p'_1]\!]$, $[\![p_1 \,\&\, p_2]\!] \equiv [\![p'_1 \,\&\, p'_2]\!]$, $[\![p_1 \parallel p_2]\!] \equiv [\![p'_1 \parallel p'_2]\!]$ and $[\![\mathcal{C}(p_1, \ldots, p_n)]\!] \equiv [\![\mathcal{C}(p'_1, \ldots, p'_n)]\!]$.*

We can use the definition of semantic equivalence to prove that the following algebraic laws hold for patterns:

▶ **Theorem 8** (Algebraic Equivalences of Patterns, I,). *For all patterns p, q, r, the following equivalences hold:*

$$\begin{array}{rclrcll} [\![p \,\&\, q]\!] & \equiv & [\![q \,\&\, p]\!] & [\![p \parallel q]\!] & \equiv & [\![q \parallel p]\!] & \text{Commutativity} \\ [\![p \,\&\, (q \,\&\, r)]\!] & \equiv & [\![(p \,\&\, q) \,\&\, r]\!] & [\![p \parallel (q \parallel r)]\!] & \equiv & [\![(p \parallel q) \parallel r]\!] & \text{Associativity} \\ [\![p \,\&\, _]\!] & \equiv & [\![p]\!] & [\![p \parallel \#]\!] & \equiv & [\![p]\!] & \text{Neutral Elements} \\ [\![\neg_]\!] & \equiv & [\![\#]\!] & [\![\neg\#]\!] & \equiv & [\![_]\!] & \text{Duality} \\ [\![\neg(p \parallel q)]\!] & \equiv & [\![(\neg p) \,\&\, (\neg q)]\!] & [\![\neg(p \,\&\, q)]\!] & \equiv & [\![(\neg p) \parallel (\neg q)]\!] & \text{De Morgan} \\ & & & [\![\neg\neg p]\!] \equiv [\![p]\!] & & & \text{Double Negation} \end{array}$$

In addition to these boolean laws we can also prove the following equivalences which involve constructor patterns:

▶ **Theorem 9** (Equivalences of Constructor Patterns, I,). *For all patterns p_1 to p_n and p'_1 to p'_n, the following equivalences hold:*

$$\begin{aligned} [\![\mathcal{C}(p_1, \ldots, p_n) \,\&\, \mathcal{C}(p'_1, \ldots, p'_n)]\!] &\equiv [\![\mathcal{C}(p_1 \,\&\, p'_1, \ldots, p_n \,\&\, p'_n)]\!] \\ [\![\mathcal{C}(p_1, \ldots, p_i \parallel p'_i, \ldots, p_n)]\!] &\equiv [\![\mathcal{C}(p_1, \ldots, p_i, \ldots, p_n) \parallel \mathcal{C}(p_1, \ldots, p'_i, \ldots, p_n)]\!] \end{aligned}$$

There are still some equivalences that we expect to hold but which are missing from Theorem 8 and Theorem 9. These additional laws, like the distributive law for and- and or-patterns, are not universally valid and require additional restrictions on patterns. We will motivate and introduce these additional constraints in the next subsection.

3.2 Linear Patterns

The first restriction we impose on patterns is *linearity*. The linearity restriction is easy to state if the syntax of patterns does not include and-patterns, or-patterns and negation-patterns: A pattern is linear if every variable occurs at most once. To see why this restriction is necessary consider the nonlinear pattern $\texttt{Cons}(x, x)$, for which we can derive the following judgment:

$$\texttt{Cons}(x, x) \triangleright \texttt{Cons}(2, \texttt{Nil}) \rightsquigarrow [x \mapsto 2, x \mapsto \texttt{Nil}] \tag{6}$$

The problem here is that multiple values are bound to the same variable, i.e. the result of the pattern match is not a proper substitution.

We have a more expressive language of algebraic patterns, and the simple definition of linearity described above is no longer sufficient. To see why this is the case, consider the pattern $(x \mathbin{\&} \texttt{True}) \parallel \texttt{False}$ that is linear according to the simple definition, but which shows the following pathological behavior:

$$(x \mathbin{\&} \texttt{True}) \parallel \texttt{False} \triangleright \texttt{True} \rightsquigarrow [x \mapsto \texttt{True}] \tag{7}$$

$$(x \mathbin{\&} \texttt{True}) \parallel \texttt{False} \triangleright \texttt{False} \rightsquigarrow [] \tag{8}$$

Equations (7) and (8) show that matching against this pattern can produce substitutions that do not have the same domain.

The two examples above tell us what the correct notion of linearity should guarantee: First, if a linear pattern matches against a value, then the resulting substitution should be a proper substitution which contains at most one mapping for a variable. Second, if we use a pattern in a clause we should be able to read off the variables that will be in the domain of the substitution of a successful match.

In order to ensure that these properties hold we have to use two mutually recursive judgments $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$, whose rules are given in Figure 2. We use $p\ \mathbf{lin}^\pm$ for patterns which satisfy both $p\ \mathbf{lin}^+$ and $p\ \mathbf{lin}^-$. Theorems 10 and 11 show that linear patterns do have the properties described in the previous paragraph.

▶ **Theorem 10** (Linear patterns produce covering substitutions, 🐓). *For any pattern p, value v and substitution σ:*
1. *If $p\ \mathbf{lin}^+$ and $p \triangleright v \rightsquigarrow \sigma$, then the domain of σ is equal to $\mathrm{FV}^e(p)$.*
2. *If $p\ \mathbf{lin}^-$ and $p \not\triangleright v \rightsquigarrow \sigma$, then the domain of σ is equal to $\mathrm{FV}^o(p)$.*

▶ **Theorem 11** (Linear patterns produce proper substitutions, 🐓). *For any pattern p, value v and substitution σ:*
1. *If $p\ \mathbf{lin}^+$ and $p \triangleright v \rightsquigarrow \sigma$, then σ is a proper substitution.*
2. *If $p\ \mathbf{lin}^-$ and $p \not\triangleright v \rightsquigarrow \sigma$, then σ is a proper substitution.*

A substitution σ is proper if any variable occurs at most once in the domain of σ.

We will now motivate why the rules in Figure 2 have the specific form that they do. Theorem 10 tells us that we should think about the variables in $\mathrm{FV}^e(p)$ as the actual pattern variables which we can use in the right-hand side of a clause, and that we should think of the variables in $\mathrm{FV}^o(p)$ as the "temporarily deactivated" pattern variables "reactivated" by negation.

Let us first look at the restrictions on the variables in $\mathrm{FV}^e(p)$, since those are more intuitive. In the or-pattern $p_1 \parallel p_2$ we require that $\mathrm{FV}^e(p_1)$ is equal to $\mathrm{FV}^e(p_2)$ since we do not know which of the two patterns will match, and we have to ensure that the same variables appear in the substitution. In the and-pattern $p_1 \mathbin{\&} p_2$ we require that $\mathrm{FV}^e(p_1)$ is

disjoint from $\text{FV}^e(p_2)$ since we do not want the same variable to be mapped to two values in the substitution. The reasoning is very similar for constructor patterns $\mathcal{C}(p_1, \ldots, p_i)$ where we require all the $\text{FV}^e(p_i)$ to be disjoint.

Next, let us look at the restrictions on $\text{FV}^o(e)$, i.e. the variables which occur under an odd number of negations. The rules for and-patterns and or-patterns are motivated by duality. We have seen that the De Morgan rules are valid for patterns. We therefore want, for example, that if the pattern $\neg(p_1 \parallel p_2)$ is linear, the pattern $\neg p_1 \,\&\, \neg p_2$ should be linear as well. For this to be true, the restrictions on $\text{FV}^o(p)$ for or-patterns have to mirror the restrictions on $\text{FV}^e(p)$ for and-patterns, and vice-versa. We still have to explain why we require for a constructor pattern $\mathcal{C}(p_1, \ldots, p_n)$ that all $\text{FV}^o(p_i)$ have to be the empty set. The reason for this restriction lies in the following semantic equivalence, which is motivated by its importance for rewriting patterns into a normal form (see Section 5.1) and which we will prove below.

$$[\![\, \neg\mathcal{C}(p_1, \ldots, p_n) \,]\!] \equiv [\![\, \neg\mathcal{C}(_, \ldots, _) \parallel \mathcal{C}(\neg p_1, \ldots, _) \parallel \ldots \parallel \mathcal{C}(_, \ldots, \neg p_n) \,]\!]$$

The pattern on the right consists of several patterns that are joined by or-patterns. The rules for or-patterns require that each disjunct contains the same sets $\text{FV}^e(-)$. But since the first pattern $\neg\mathcal{C}(_, \ldots, _)$ doesn't contain any variables, the other disjuncts must not contain any variables under an even number of negations. And since the subpatterns p_i occur under a negation, we can deduce that the patterns p_i must not contain any variables under an odd number of negations.

Using linearity we can now prove the missing equivalences which were not included in Theorems 8 and 9.

▶ **Theorem 12** (Algebraic equivalences of Patterns, II, 🐓)**.** *For all patterns p, q and r, the following equivalences hold if the patterns on both sides are linear (i.e. $\mathbf{lin}^{\pm}$).*

$[\![\, p \,\&\, (q \parallel r) \,]\!] \equiv [\![\, (p \,\&\, q) \parallel (p \,\&\, r) \,]\!]$	$[\![\, p \parallel (q \,\&\, r) \,]\!] \equiv [\![\, (p \parallel q) \,\&\, (p \parallel r) \,]\!]$	Distributivity
$[\![\, p \,\&\, p \,]\!] \equiv [\![\, p \,]\!]$	$[\![\, p \parallel p \,]\!] \equiv [\![\, p \,]\!]$	Idempotence
$[\![\, p \,\&\, \# \,]\!] \equiv [\![\, \# \,]\!]$	$[\![\, p \parallel _ \,]\!] \equiv [\![\, _ \,]\!]$	Zeros

Theorem 8 and Theorem 12 taken together show that patterns form a boolean algebra, with the caveat that the conjunction or disjunction of two linear patterns does not necessarily result in a linear pattern. We can also prove some additional equivalences involving constructor patterns.

▶ **Theorem 13** (Equivalences of Constructor Patterns, II, 🐓)**.** *For all patterns p_1 to p_n and p'_1 to p'_n and constructors $\mathcal{C} \neq \mathcal{C}'$, the following equivalences hold if the patterns on both sides are linear (i.e. $\mathbf{lin}^{\pm}$).*

$$\begin{aligned}
[\![\, \mathcal{C}(p_1, \ldots, \#, \ldots, p_n) \,]\!] &\equiv [\![\, \# \,]\!] \\
[\![\, \mathcal{C}(p_1, \ldots, p_n) \,\&\, \mathcal{C}'(p'_1, \ldots, p'_m) \,]\!] &\equiv [\![\, \# \,]\!] \\
[\![\, \mathcal{C}(p_1, \ldots, p_n) \,\&\, \neg\mathcal{C}'(p'_1, \ldots, p'_m) \,]\!] &\equiv [\![\, \mathcal{C}(p_1, \ldots, p_n) \,]\!] \\
[\![\, \neg\mathcal{C}(p_1, \ldots, p_n) \,]\!] &\equiv [\![\, \neg\mathcal{C}(_, \ldots, _) \parallel \mathcal{C}(\neg p_1, \ldots, p_n) \parallel \ldots \parallel \mathcal{C}(p_1, \ldots, \neg p_n) \,]\!]
\end{aligned}$$

The equivalences of Theorem 13 are essential for proving the correctness of the normalization algorithm specified in Section 5.

3.3 Deterministic Patterns

We also have to introduce a notion of deterministic patterns. To see why this is necessary, consider the pathological pattern $\texttt{Pair}(x, _) \,\|\, \texttt{Pair}(_, x)$. This pattern is linear, but when we match against the value $\texttt{Pair}(2, 4)$ we can either bind the value 2 or the value 4 to the variable x. Deterministic patterns guarantee that when there are multiple derivations which show that a pattern matches against a value, then these alternatives must yield equivalent substitutions[3].

To define deterministic patterns we first have to introduce overlapping and disjoint patterns. In the extended version [2] we give a procedure for deciding whether two patterns in a certain normalized form – which we will introduce in Section 5 – overlap.

▶ **Definition 14** (Overlapping and Disjoint Patterns). *Two patterns p and q are overlapping, written $p \between q$, if there exists a value v and substitutions σ_1 and σ_2 such that $p \triangleright v \rightsquigarrow \sigma_1$ and $q \triangleright v \rightsquigarrow \sigma_2$. Two patterns p and q are disjoint, written $p \not\between q$ if such a value and such substitutions do not exist.*

We write p **det** for patterns which satisfy the determinism restriction specified in Figure 2. Note that there are two rules for or-patterns: The rule D-Or$_1$ says that an or-pattern is deterministic if the subpatterns are disjoint, whereas the rule D-Or$_2$ allows arbitrary deterministic subpatterns as long as they do not contain free variables. As for the linearity restriction, these two rules for or-patterns give rise to corresponding rules for and-patterns via De Morgan duality: Given that the pattern $\neg(p1 \,\|\, p2)$ is deterministic, we want the semantically equivalent pattern $\neg p_1 \,\&\, \neg p_2$ to be deterministic as well.

▶ **Theorem 15** (Matching for wellformed patterns is deterministic, 🐓). *For any pattern p, value v and substitutions σ_1, σ_2:*

1. *If p $\mathbf{lin}^+$, p **det** and $p \triangleright v \rightsquigarrow \sigma_1$ and $p \triangleright v \rightsquigarrow \sigma_2$, then $\sigma_1 \equiv \sigma_2$.*
2. *If p $\mathbf{lin}^-$, p **det** and $p \not\triangleright v \rightsquigarrow \sigma_1$, and $p \not\triangleright v \rightsquigarrow \sigma_2$, then $\sigma_1 \equiv \sigma_2$.*

This property is also necessary to show that the evaluation of terms is deterministic, since we use the relation $p \triangleright v \rightsquigarrow \sigma$ when we define the evaluation of pattern matches in the next subsection.

4 Terms and Types

In this section we present the second part of our solution: default clauses. We present a small language, specify its operational semantics and introduce the typing rules for patterns and expressions. The language contains only data types and pattern matching and omits higher-order functions, but these could be added in the standard way.

4.1 The Syntax of Expressions

Expressions e consist of variables x, the application of a constructor to arguments $\mathcal{C}(e_1, \ldots, e_n)$ and case expressions. Every case expression consists of a scrutinee, a list of normal clauses c which all have the form $p \Rightarrow e$, and a default clause **default** $\Rightarrow e$.

$$\begin{array}{lll} e & ::= & x \mid \mathcal{C}(e_1, \ldots, e_n) \mid \textbf{case}\ e\ \textbf{of}\ \{\ c_1, \ldots, c_n\ ;\ \textbf{default} \Rightarrow e\ \} \quad \textit{Expression} \\ c & ::= & p \Rightarrow e \quad \textit{Clause} \end{array}$$

[3] Most languages which implement or-patterns do not check whether patterns are deterministic and instead evaluate or-patterns left-to-right, but this invalidates algebraic laws such as the commutativity of or-patterns. If we need such left-biased or-patterns $p \,\|^{\leftarrow}\, q$ then we can easily define them as $p \,\|\, (\neg p \,\&\, q)$.

Every case expression contains exactly one default clause, but a realistic language would also allow to omit the default clause if all other clauses are exhaustive. As we already mentioned, **default** does not belong to the syntax of patterns, but to the syntax of case expressions. In particular, the meaning of a default clause is not equal to the meaning of a clause $_ \Rightarrow e_d$ which uses a wildcard pattern, but equal to a clause $\neg p_1 \,\&\, \cdots \,\&\, \neg p_n \Rightarrow e_d$, where the p_i are all the patterns from the other clauses of the case expression. For this reason it is not possible to define the meaning of a default clause in isolation, unlike for normal clauses, and a default clause also does not immediately give rise to a valid equation for term rewriting.

4.2 Operational Semantics

We formalize a call-by-value semantics in which the only values v are constructors applied to other values. We use evaluation contexts E and a congruence rule E-Cong to specify evaluation within a subterm, following standard practice [9].

$$E \quad ::= \quad \square \mid \mathcal{C}(v_1, \ldots, v_n, E, e_1, \ldots, e_m) \mid \textbf{case } E \textbf{ of } \{\ c_1, \ldots, c_n\,;\, \textbf{default} \Rightarrow e\ \}$$

We write $e \to^r e'$ for the single-step reduction of the expression e to e' which does not use the congruence rule, and $e \to e'$ for the single step reduction within an evaluation context. The evaluation of case expressions is governed by the two rules E-Match and E-Default. The rule E-Match allows to reduce a case expression using any of the normal clauses that match the scrutinee; in particular, it is not necessary to try the clauses in the order in which they occur. On the right-hand side of that rule we write $e\,\sigma$ for the result of applying the substitution σ to the expression e. The rule E-Default allows to reduce a case expression to the expression of the default clause if none of the other clauses match against the scrutinee.

Single-step evaluation: $e \to e$

$$\frac{p \rhd v \rightsquigarrow \sigma}{\textbf{case } v \textbf{ of } \{c_1, \ldots, c_n, p \Rightarrow e, c_{n+2}, \ldots, c_m\,;\, \textbf{default} \Rightarrow e_d\} \to^r e\sigma}\ \text{E-Match}$$

$$\frac{p_1 \not\rhd v \rightsquigarrow \sigma_1 \quad \cdots \quad p_n \not\rhd v \rightsquigarrow \sigma_n}{\textbf{case } v \textbf{ of } \{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\,;\, \textbf{default} \Rightarrow e_d\ \} \to^r e_d}\ \text{E-Default}$$

$$\frac{e \to^r e'}{E[e] \to E[e']}\ \text{E-Cong}$$

Nothing in these rules guarantees that evaluation is deterministic. For example, we can write a case expression with overlapping patterns. If we want evaluation to be deterministic, then we have to check that expressions are wellformed, written as e ✓. For this we have to check that all patterns which occur in case expressions are deterministic, positively linear, and that no two clauses in a case expression have overlapping patterns.

Wellformed expressions: e ✓

$$\frac{}{x\ \checkmark}\ \text{Wf-Var} \qquad\qquad \frac{e_1 \checkmark \cdots e_n \checkmark}{\mathcal{C}(e_1, \ldots, e_n)\ \checkmark}\ \text{Wf-Ctor}$$

$$\frac{\{e, e_d, e_1, \ldots, e_n\}\ \checkmark \qquad p_i\ \textbf{det} \quad p_i\ \textbf{lin}^+ \quad \text{if } i \neq j \text{ then } p_i \not\between p_j}{\textbf{case } e \textbf{ of } \{\ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n\,;\, \textbf{default} \Rightarrow e_d\ \}\ \checkmark}\ \text{Wf-Case}$$

The evaluation of wellformed expressions is deterministic. Furthermore, to check whether an expression is wellformed does not depend on any type information.

▶ **Theorem 16** (Evaluation of wellformed expressions is deterministic). *For all expressions e which are wellformed (that is e ✓ holds), evaluation is deterministic: If $e \to e_1$ and $e \to e_2$, then $e_1 = e_2$.*

Multi-step evaluation $\to^*$ is defined as the reflexive-transitive closure of single-step evaluation and allows us to define when two closed expressions are semantically equivalent:

▶ **Definition 17** (Semantic Equivalence of Expressions). *Two closed expressions e and e' are semantically equivalent, written $[\![e]\!] \equiv [\![e']\!]$, if either both of them evaluate to the same value v, i.e. $e \to^* v$ and $e' \to^* v$, or if both expressions diverge.*

▶ **Theorem 18** (Permutation of clauses does not change semantics). *The meaning of a wellformed pattern match does not change when we permute clauses. If $\overline{c'}$ is a permutation of the list of clauses $\overline{c}$, then $[\![\textbf{case}\ v\ \textbf{of}\ \{\overline{c}\ ;\textbf{default} \Rightarrow e_d\}]\!] \equiv [\![\textbf{case}\ v\ \textbf{of}\ \{\overline{c'}\ ;\textbf{default} \Rightarrow e_d\}]\!]$.*

This theorem makes formal what we call pure no-overlap semantics for pattern matching. The next theorem guarantees that we can use algebraic principles when reasoning about patterns in clauses.

▶ **Theorem 19** (Substitution of equivalent patterns preserves semantics). *If $[\![p_1]\!] \equiv [\![p_2]\!]$, then*

$$[\![\textbf{case}\ v\ \textbf{of}\ \{p_1 \Rightarrow e, \overline{c}\ ;\textbf{default} \Rightarrow e_d\}]\!] \equiv [\![\textbf{case}\ v\ \textbf{of}\ \{p_2 \Rightarrow e, \overline{c}\ ;\textbf{default} \Rightarrow e_d\}]\!].$$

4.3 Typing Rules

In this section, we are going to introduce types, typing contexts and (pattern) typing rules. In order to illustrate the typing rules we include three simple data types: the boolean type $\mathbb{B}$ with constructors `True` and `False`, the type of pairs $\tau \otimes \tau$ with the constructor `Pair`, and the type of binary sums $\tau \oplus \tau$ with constructors ι_1 and ι_2.

$$\begin{array}{lll} \tau & ::= & \mathbb{B} \mid \tau \otimes \tau \mid \tau \oplus \tau & \textit{Types} \\ \Gamma, \Delta & ::= & \cdot \mid \Gamma, x : \tau & \textit{Typing contexts} \end{array}$$

We use the judgment $\Gamma; \Delta \Rightarrow p : \tau$ to express that the pattern p matches against a value of type τ and binds variables whose types are recorded in the context Γ on a successful match, and in context Δ in the case of a non-successful match. In other words, the typing context Γ is used for typing the pattern variables that occur under an even number of negations, and the typing context Δ is used for typing the variables under an odd number of negations. We write $\Gamma_1 +\!\!+ \Gamma_2$ for the naive concatenation of two typing contexts.

Pattern typing: $\Gamma; \Delta \Rightarrow p : \tau$

$$\frac{}{x : \tau; \cdot \Rightarrow x : \tau}\ \text{P-VAR} \qquad \frac{}{\cdot; \cdot \Rightarrow _ : \tau}\ \text{P-WILDCARD} \qquad \frac{}{\cdot; \cdot \Rightarrow \# : \tau}\ \text{P-ABSURD}$$

$$\frac{\Gamma_1; \Delta \Rightarrow p_1 : \tau \quad \Gamma_2; \Delta \Rightarrow p_2 : \tau}{\Gamma_1 +\!\!+ \Gamma_2; \Delta \Rightarrow p_1 \,\&\, p_2 : \tau}\ \text{P-AND} \qquad \frac{\Gamma; \Delta_1 \Rightarrow p_1 : \tau \quad \Gamma; \Delta_2 \Rightarrow p_2 : \tau}{\Gamma; \Delta_1 +\!\!+ \Delta_2 \Rightarrow p_1 \parallel p_2 : \tau}\ \text{P-OR}$$

$$\frac{\Gamma; \Delta \Rightarrow p : \tau}{\Delta; \Gamma \Rightarrow \neg p : \tau}\ \text{P-NEG} \qquad \frac{}{\cdot; \cdot \Rightarrow \texttt{True} : \mathbb{B}}\ \text{P-TRUE} \qquad \frac{}{\cdot; \cdot \Rightarrow \texttt{False} : \mathbb{B}}\ \text{P-FALSE}$$

$$\frac{\Gamma; \Delta \Rightarrow p : \tau_1}{\Gamma; \Delta \Rightarrow \iota_1(p) : \tau_1 \oplus \tau_2}\ \text{P-INL} \qquad \frac{\Gamma; \Delta \Rightarrow p : \tau_2}{\Gamma; \Delta \Rightarrow \iota_2(p) : \tau_1 \oplus \tau_2}\ \text{P-INR}$$

$$\frac{\Gamma_1; \Delta_1 \Rightarrow p_1 : \tau_1 \qquad \Gamma_2; \Delta_2 \Rightarrow p_2 : \tau_2}{\Gamma_1 +\!\!+ \Gamma_2; \Delta_1 +\!\!+ \Delta_2 \Rightarrow \texttt{Pair}(p_1, p_2) : \tau_1 \otimes \tau_2} \text{ P-PAIR}$$

These pattern typing rules do not make sense on their own, but only in conjunction with the rules for linearity. For example, the typing rules allow to derive the typing judgment $x : \tau_1, x : \tau_2; \cdot \Rightarrow \texttt{Pair}(x, x) : \tau_1 \otimes \tau_2$, but the linearity rules disallow this pattern[4]. Lastly, we present the rules for typing expressions:

Expression typing: $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e : \sigma \qquad \Gamma \vdash e_d : \tau \qquad \forall i: \; \Gamma_i; \Delta \Rightarrow p_i : \sigma \text{ and } \Gamma +\!\!+ \Gamma_i \vdash e_i : \tau}{\Gamma \vdash \textbf{case } e \textbf{ of } \{ \; p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n \; ; \textbf{default} \Rightarrow e_d \; \} : \tau} \text{ T-MATCH}$$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \text{ T-VAR} \qquad \frac{}{\Gamma \vdash \texttt{True} : \mathbb{B}} \text{ T-TRUE} \qquad \frac{}{\Gamma \vdash \texttt{False} : \mathbb{B}} \text{ T-FALSE}$$

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{Pair}(e_1, e_2) : \sigma \otimes \tau} \text{ T-PAIR} \quad \frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \iota_1(e) : \sigma \oplus \tau} \text{ T-INL} \quad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \iota_2(e) : \sigma \oplus \tau} \text{ T-INR}$$

In the rule T-MATCH we use the pattern typing rule for the pattern in each clause, and add the resulting typing context Γ_i which contains the types of all pattern variables under an even number of negations to the outer typing context Γ in the respective clause.

5 Normalizing Algebraic Patterns

So far, we have considered algebraic patterns in their unrestricted form which is written by the programmer and checked by the compiler. In this section, we consider normal forms for patterns which simplify the compilation algorithm in Section 6. We end up with a normal form that is a variant of disjunctive normal forms, but which takes the use of constructors and variables into account. We describe a normalization procedure which proceeds in three steps: The first step, described in Section 5.1, pushes negation-patterns inward so that they only occur applied to constructors without subpatterns, or to variables. The second step, described in Section 5.2, pushes or-patterns outwards, so that we end up with a disjunctive normal form, that is a disjunction of elementary conjuncts. In the third step, we further simplify these elementary conjuncts; this is described in Section 5.3. The default clause is not affected by the normalization procedure described in this section; the compilation of case expressions to decision trees described in the next section can handle default clauses natively.

5.1 Computing the Negation Normal Form

Consider the pattern $\neg$`Pair(True, False)` which matches anything except a tuple consisting of the boolean values `True` and `False`. Another way to express the behavior of this pattern is to say that it matches anything which isn't a tuple, or it matches a tuple if either the left element isn't the value `True` or the right element isn't the value `False`. We can therefore also write the pattern as $\neg\texttt{Pair}(_, _) \parallel \texttt{Pair}(\neg\texttt{True}, _) \parallel \texttt{Pair}(_, \neg\texttt{False})$. We can generalize this observation and use it to push all negation-patterns inside, by repeatedly rewriting the pattern using the last equation of Theorem 13. After we have normalized patterns in this way, negation-patterns only occur applied to variables or to constructors where all subpatterns of the constructor are wildcard patterns. We abbreviate these negated constructor patterns $\neg\mathcal{C}^n(_, \ldots, _)$ as $\neg\mathcal{C}^n$.

[4] An implementation might choose to check linearity and typing of patterns simultaneously, but we prefer to present them separately to illustrate that we can also use algebraic patterns in the untyped setting.

▶ **Definition 20** (Negation Normal Forms). *The syntax of negation normal forms is:*

$$N \quad ::= \quad x \mid \neg x \mid \neg \mathcal{C}^n \mid \mathcal{C}^n(N_1, \ldots, N_n) \mid N \,\&\, N \mid N \parallel N \mid _ \mid \#$$

$$\begin{array}{rlrl}
\texttt{nnf}^-(p_1 \,\&\, p_2) & := \texttt{nnf}^-(p_1) \parallel \texttt{nnf}^-(p_2) & \texttt{nnf}^+(p_1 \,\&\, p_2) & := \texttt{nnf}^+(p_1) \,\&\, \texttt{nnf}^+(p_2) \\
\texttt{nnf}^-(p_1 \parallel p_2) & := \texttt{nnf}^-(p_1) \,\&\, \texttt{nnf}^-(p_2) & \texttt{nnf}^+(p_1 \parallel p_2) & := \texttt{nnf}^+(p_1) \parallel \texttt{nnf}^+(p_2) \\
\texttt{nnf}^-(\neg p) & := \texttt{nnf}^+(p) & \texttt{nnf}^+(\neg p) & := \texttt{nnf}^-(p) \\
\texttt{nnf}^-(\#) & := _ & \texttt{nnf}^+(\#) & := \# \\
\texttt{nnf}^-(_) & := \# & \texttt{nnf}^+(_) & := _ \\
\texttt{nnf}^-(x) & := \neg x & \texttt{nnf}^+(x) & := x
\end{array}$$

$$\begin{array}{rl}
\texttt{nnf}^-(\mathcal{C}^n(p_1, \ldots, p_n)) & := \neg \mathcal{C}^n \parallel \mathcal{C}^n(\texttt{nnf}^-(p_1), \ldots, _) \parallel \ldots \parallel \mathcal{C}^n(_, \ldots, \texttt{nnf}^-(p_n)) \\
\texttt{nnf}^+(\mathcal{C}^n(p_1, \ldots, p_n)) & := \mathcal{C}^n(\texttt{nnf}^+(p_1), \ldots, \texttt{nnf}^+(p_n))
\end{array}$$

The normalization procedure $\texttt{nnf}(-)$ is split into the two functions $\texttt{nnf}^-(-)$ and $\texttt{nnf}^+(-)$ which track if we are currently normalizing a negated or a non-negated pattern. At the top-level, the function $\texttt{nnf}(-)$ therefore just invokes $\texttt{nnf}^+(-)$.

▶ **Example 21.** We compute the negation normal form of our running example.

$$\texttt{nnf}^+(x \,\&\, (\texttt{Sa} \parallel \texttt{Su})) = x \,\&\, (\texttt{Sa} \parallel \texttt{Su}) \qquad \texttt{nnf}^+(x \,\&\, \neg(\texttt{Sa} \parallel \texttt{Su})) = x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})$$

We treat the negation normal forms N as a subset of patterns p, and hence all functions and properties of patterns can also be applied to normal forms. If we translate a pattern into negation normal form then the pattern is still wellformed, can be typed using the same context and type, and has the same semantics as before. This is witnessed by the following lemma.

▶ **Lemma 22** (Negation Normalization Preserves Linearity, Typing and Semantics). *For all patterns p, contexts Γ, Δ and types τ, we have:*

1. *Linearity is preserved: If p $\mathbf{lin}^+$, then $\texttt{nnf}(p)$ $\mathbf{lin}^+$.*
2. *Typing is preserved: If $\Gamma; \Delta \Rightarrow p : \tau$, then $\Gamma; \Delta \Rightarrow \texttt{nnf}(p) : \tau$.*
3. *The patterns are semantically equivalent: If p $\mathbf{lin}^\pm$, then $[\![p]\!] \equiv [\![\texttt{nnf}(p)]\!]$.*

Proof. Available in the appendix of the extended version of this paper [2]. ◀

5.2 Computing the Disjunctive Normal Form

The next step is to bring the negation normal form into a disjunctive normal form. Patterns in disjunctive normal form consist of an outer disjunction of so-called "elementary conjuncts" which do not contain or-patterns. The syntax of the disjunctive normal forms D and elementary conjuncts K is given below. We can normalize a pattern in negation normal form to the disjunctive normal form by the function $\texttt{dnf}(-)$ which takes a pattern in negation normal form and returns a set of elementary conjuncts. At the top-level we have to wrap the result in a $\parallel \{\ldots\}$ node.

▶ **Definition 23** (Disjunctive Normal Form).

$$\begin{array}{rcll}
D & ::= & \parallel \{K_1, \ldots, K_n\} & \text{Disjunctive Normal Form} \\
K & ::= & x \mid \neg x \mid \neg \mathcal{C}^n \mid \mathcal{C}^n(K_1, \ldots, K_n) \mid K \,\&\, K \mid _ \mid \# & \text{Elementary Conjunct}
\end{array}$$

$$\texttt{dnf}(x) := \{x\} \quad \texttt{dnf}(\neg x) := \{\neg x\} \quad \texttt{dnf}(\neg \mathcal{C}^n) := \{\neg \mathcal{C}^n\} \quad \texttt{dnf}(_) := \{_\} \quad \texttt{dnf}(\#) := \{\#\}$$

$$\begin{array}{rcl}
\texttt{dnf}(\mathcal{C}^n(N_1, \ldots, N_n)) & := & \{\mathcal{C}(k_1, \ldots, k_n) \mid k_1 \in \texttt{dnf}(N_1), \ldots, k_n \in \texttt{dnf}(N_n)\} \\
\texttt{dnf}(N_1 \,\&\, N_2) & := & \{k_1 \,\&\, k_2 \mid k_1 \in \texttt{dnf}(N_1), k_2 \in \texttt{dnf}(N_2)\} \\
\texttt{dnf}(N_1 \parallel N_2) & := & \texttt{dnf}(N_1) \cup \texttt{dnf}(N_2)
\end{array}$$

▶ **Example 24.** We compute the disjunctive normal form for our running example.

$$\texttt{dnf}(x \,\&\, (\texttt{Sa} \parallel \texttt{Su})) = \parallel \{x \,\&\, \texttt{Sa}, x \,\&\, \texttt{Su}\} \qquad \texttt{dnf}(x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})) = \parallel \{x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})\}$$

We can embed the disjunctive normal form $\parallel \{K_1, \ldots, K_n\}$ into patterns as $K_1 \parallel \ldots \parallel K_n$. This set of conjuncts is by design never empty ($n > 0$), however, if we do admit this case ($n = 0$) it translates to the absurd-pattern # and is treated as such in any use case (see Section 6).

5.3 Normalizing Elementary Conjuncts

As a last step, we further simplify the elementary conjuncts of Section 5.2. We write $\overline{D}$ and $\overline{K}$ for these normalized disjunctive normal forms and normalized elementary conjuncts:

▶ **Definition 25** (Normalized Disjunctive Normal Form).

$$\begin{array}{lcl} \overline{D} & ::= & \parallel \{\overline{K}_1, \ldots, \overline{K}_n\} \\ \overline{K} & ::= & \{x_1, \ldots, x_n\} \,\&\, \mathcal{C}(\overline{K}_1, \ldots, \overline{K}_m) \mid \{x_1, \ldots, x_n\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_m\} \mid \{x_1, \ldots, x_n\} \,\&\, \# \end{array}$$

These normalized conjuncts follow from the observation that there are essentially only three different kinds of conjuncts: positive, negative and unsatisfiable. Each of these three conjuncts can bind a set of variables $\{x_1, \ldots, x_n\}$. A *positive conjunct* $\{x_1, \ldots, x_n\} \,\&\, \mathcal{C}(\overline{K}_1, \ldots, \overline{K}_m)$ matches against any value headed by the constructor $\mathcal{C}$, and binds that value against all the variables in the set. A *negative conjunct* $\{x_1, \ldots, x_n\} \,\&\, \neg\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$ matches against any value which is headed by a constructor which is *not* in the list $\{\mathcal{C}_1, \ldots, \mathcal{C}_m\}$. The negative conjunct $\{x\} \& \neg\{\}$ can be used to represent a single pattern variable x, and similarly $\{\} \& \neg\{\}$ can be used to represent a wildcard pattern. An *unsatisfiable conjunct* $\{x_1, \ldots, x_n\} \,\&\, \#$ never matches against a value.

Computing these normalized conjuncts consists in collecting all the variables, as well as all negated and non-negated head constructors of an elementary conjunct. Intuitively, the head constructors are constructors that occur in a pattern without being nested in a subpattern of another constructor. An elementary conjunct is unsatisfiable if two different non-negated constructors appear in the conjunct, or if a head constructor appears both negated and non-negated. An elementary conjunct can be simplified to a negative conjunct if only negated head constructors appear in the original conjunct. Lastly, an elementary conjunct can be simplified to a positive conjunct if neither of the two cases above apply. We write $\texttt{norm}(-)$ for the function which computes these normalized conjuncts; the formal definition of this function is given in the appendix of the extended version of this paper [2]. We also remove negated variables in this step and treat them like absurd patterns; this is, strictly speaking, not semantics preserving, but negated variables cannot be used in the right-hand side of a clause anyway, so this has no effect on the compilation of programs.

▶ **Example 26.** We compute the normalized disjunctive normal form for our running example.

$$\begin{aligned} \texttt{norm}(\parallel \{x \,\&\, \texttt{Sa}, x \,\&\, \texttt{Su}\}) &= \parallel \{\{x\} \,\&\, \texttt{Sa}, \{x\} \,\&\, \texttt{Su}\} \\ \texttt{norm}(\parallel \{x \,\&\, (\neg\texttt{Sa}) \,\&\, (\neg\texttt{Su})\}) &= \parallel \{\{x\} \,\&\, \neg\{\texttt{Sa}, \texttt{Su}\}\} \end{aligned}$$

Like in Section 5.2, we treat the normal forms $\overline{D}$ and $\overline{K}$ as subsets of patterns p. We use patterns in normalized disjunctive normal form in the next section where we define a pattern matching compilation algorithm for nested patterns.

6 Compiling Algebraic Patterns

We have shown that algebraic patterns are more expressive because they explicitly allow us to speak about the complement of a pattern. But programmers won't use them if they have to pay for the gain in expressivity with slower programs. In order to compile programs using these patterns to efficient code we therefore have to use and modify one of the available pattern matching compilation algorithms.

These algorithms can be divided into two general classes: Algorithms which use backtracking [1] and algorithms which compile to decision trees [23]. The algorithms which use backtracking guarantee that the size of the generated code is linear in the size of the input, but the generated code might perform redundant tests. Algorithms which compile to decision trees never examine scrutinees more than once but the generated code might be exponential in the size of the input. Both classes of algorithms are competitive when they are optimized [20, 23], so it is not possible to say which alternative is preferable in general.

In this article we modify and extend the compilation scheme laid out by Maranget [23] which is based on decision trees. His algorithm already includes a treatment of or-patterns, and we will extend it to the normalized patterns that we introduced in Section 5. We introduce the general idea of the compilation to decision trees in Section 6.1, present the central branching step in Section 6.2 in detail and illustrate the algorithm with an example in Section 6.3. We conclude with a statement on the correctness of our extended compilation scheme in Section 6.4.

6.1 General Idea of the Algorithm

The algorithm operates on a generalization of the case expression $\textbf{case } v \textbf{ of } \{ p_1 \Rightarrow e_1, \ldots, p_n \Rightarrow e_n; \textbf{default} \Rightarrow e \}$ that we introduced in Section 4.1. Instead of only one scrutinee v and one pattern p in each clause we now allow a vector $v_1, \ldots, v_n$ of scrutinees and a matrix of clauses with patterns in nDNF (see Section 5.3).

$$\textbf{case } v_1, \ldots, v_n \textbf{ of } \begin{bmatrix} \overline{D}_1^1 & \ldots & \overline{D}_n^1 & \Rightarrow e^1 \\ & \vdots & & \\ \overline{D}_1^m & \ldots & \overline{D}_n^m & \Rightarrow e^m \\ & \textbf{default} & & \Rightarrow e_d \end{bmatrix} \qquad \text{(Input)}$$

In such multi-column case expressions, the i-th pattern column tests the i-th scrutinee v_i. The main idea of the algorithm is that we take such a multi-column case expression as input and examine all scrutinees – and subscrutinees – one by one, producing a nesting of single-column case expressions with *simple* patterns as output. These simple patterns consist only of constructors applied to variables. We start the compilation by embedding an ordinary case expression into this generalized form on which we call the function *compile*. This function can take one of three steps. It either terminates with a **Default** or **Simple** step, or it examines one of the scrutinees in a **Branch** step.

Default If $\textbf{default} \Rightarrow e_d$ is the only clause then we terminate with the expression e_d.

Simple If the first row consists only of variable patterns of the form $\{x_1, \ldots x_k\} \,\&\, \neg\{\}$ then compilation terminates with the right-hand side of that clause together with appropriate substitutions. This rule also applies if $n = 0$.

Branch If neither of the above steps apply we pick a column i (with $1 \leq i \leq n$) and examine its scrutinee v_i in a single-column case against simple constructor patterns.

All the complexity of the algorithm lies within the **Branch** step, so we will discuss its details in the next subsection.

6.2 The Branching Step

The first step is to choose on which index i we want to perform the case split. This choice does not affect the correctness of the algorithm; selecting a column is a therefore a matter of optimization which other authors [26, 23] have discussed in detail. We only enforce that the selected pattern column must contain outermost constructors. Such a column must necessarily exist if neither the **Default** nor the **Simple** step apply.

Next, we have to gather the constructors against which we can test the scrutinee v_i. We collect these so-called head constructors into a set $\mathcal{H}(\overline{D}_i)$ defined below.

$$\mathcal{H}(\overline{D}_i) := \bigcup_{j=1,\dots,m} \mathtt{head}(\overline{D}_i^j) \;\; \textit{with} \;\; \begin{array}{lcl} \mathtt{head}(\| \, \{\overline{K}_1, \dots, \overline{K}_t\}) & := & \bigcup_{l=1,\dots,t} \mathtt{head}(\overline{K}_l) \\ \mathtt{head}(\{x_1, \dots, x_s\} \,\&\, \mathcal{C}(\overline{K}_1, \dots, \overline{K}_t)) & := & \{\mathcal{C}\} \\ \mathtt{head}(\{x_1, \dots, x_s\} \,\&\, \neg\{\mathcal{C}_1, \dots, \mathcal{C}_t\}) & := & \{\mathcal{C}_1, \dots, \mathcal{C}_t\} \\ \mathtt{head}(\{x_1, \dots, x_s\} \,\&\, \#) & := & \emptyset \end{array}$$

Once we have found these head constructors $\{\mathcal{C}^{n_1}, \dots, \mathcal{C}^{n_z}\}$ we generate fresh pattern variables x_1^k to $x_{n_k}^k$ for each constructor $\mathcal{C}^{n_k}$ and a simple pattern match. For each clause we have gained information about the scrutinee v_i: Within the right-hand side of a constructor clause, we know that our scrutinee must have conformed to the shape of our constructor. Likewise, if we arrive at the right-hand side of the default clause, we know that our scrutinee must have been different to all the constructors $\{\mathcal{C}^{n_1}, \dots, \mathcal{C}^{n_z}\}$ before. We use this knowledge by computing clause-specific *subproblems* of our initial Input and we do so via the function $\mathcal{S}$ for constructor clauses and function $\mathcal{D}$ for the default clause.

$$\textbf{case } v_i \textbf{ of} \begin{bmatrix} \mathcal{C}^{n_1}(x_1^1, \dots, x_{n_1}^1) & \Rightarrow & \textit{compile}(\mathcal{S}(i, \mathcal{C}^{n_1}(x_1^1, \dots, x_{n_1}^1), \text{Input})) \\ \vdots & & \vdots \\ \mathcal{C}^{n_z}(x_1^z, \dots, x_{n_z}^z) & \Rightarrow & \textit{compile}(\mathcal{S}(i, \mathcal{C}^{n_z}(x_1^z, \dots, x_{n_z}^z), \text{Input})) \\ \textbf{default} & \Rightarrow & \textit{compile}(\mathcal{D}(i, \{\mathcal{C}^{n_1}, \dots, \mathcal{C}^{n_z}\}, \text{Input})) \end{bmatrix} \quad \text{(Output)}$$

All that is left is to compute these subproblems of *specialization* and *default* which are the arguments of recursive invocations of the compile function.

6.2.1 Computing Specialization

We now show how to compute $\mathcal{S}(i, \mathcal{C}^k(x_1, \dots, x_k), \text{Input})$ which specializes Input into a constructor-specific subproblem that utilizes *the assumption that the scrutinee v_i has already matched the constructor pattern $\mathcal{C}^k(x_1, \dots, x_k)$*. We therefore remove the old scrutinee v_i and replace it with the bound variables $x_1, \dots, x_k$ as new scrutinees.

$$\mathcal{S}(i, \mathcal{C}^k(x_1, \dots, x_k), \text{Input}) \;=\; \textbf{case } \; x_1, \dots, x_k, v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n \;\textbf{ of } \; \big[S\big]$$

We now have to build a new clause matrix $\big[S\big]$ that matches against the new list of scrutinees and do so by iterating over the previous clauses. Each clause of the old Input contributes to $\big[S\big]$ only if its i-th pattern could have matched the old scrutinee v_i of shape $\mathcal{C}^k$ by *assumption*. This is, of course, the case for the default clause which we keep unchanged. Otherwise, these i-th patterns are in normalized DNF, and we have to check whether one of its elementary conjuncts could have matched. The following list gives all inclusion rules.

1. Row $\left[\textbf{default} \Rightarrow e_d\right]$ is added to $\left[S\right]$ unchanged.
2. Row $\left[\overline{D}^j_1 \ \dots \ \overline{D}^j_{i-1} \ \overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} \ \overline{D}^j_{i+1} \ \dots \ \overline{D}^j_n \ \Rightarrow \ e^j\right]$ only contributes according to $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}}$:
   - **a.** If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \{y_1, \dots y_s\} \,\&\, \mathcal{C}^k(\overline{K}_1, \dots, \overline{K}_k)$,
     then add the row $\left[\overline{K}_1 \dots \overline{K}_k \quad \overline{D}^j_1 \dots \overline{D}^j_{i-1}\overline{D}^j_{i+1} \dots \ \overline{D}^j_n \quad \Rightarrow e^j[\{y_1, \dots y_s\} \mapsto v_i]\right]$.
   - **b.** If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \{y_1, \dots y_s\} \,\&\, \neg\{\mathcal{C}_1, \dots, \mathcal{C}_t\}$ and $\mathcal{C}^k \notin \{\mathcal{C}_1, \dots, \mathcal{C}_t\}$,
     then add the row $\left[_ \ \dots \ _ \quad \overline{D}^j_1 \dots \overline{D}^j_{i-1}\overline{D}^j_{i+1} \dots \ \overline{D}^j_n \quad \Rightarrow e^j[\{y_1, \dots y_s\} \mapsto v_i]\right]$.
     (Note that we use _ simply as a shorthand for the equivalent nDNF $\{\} \,\&\, \neg\{\}$)
   - **c.** If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \|\{\overline{K}_1, \dots, \overline{K}_t\}$, then
     recursively iterate the rows of $\begin{bmatrix} \overline{D}^j_1 & \dots & \overline{D}^j_{i-1} & \mathbf{K}_1 & \overline{D}^j_{i+1} & \dots & \overline{D}^j_n & \Rightarrow & e^j \\ & & & \vdots & & & & & \\ \overline{D}^j_1 & \dots & \overline{D}^j_{i-1} & \mathbf{K}_t & \overline{D}^j_{i+1} & \dots & \overline{D}^j_n & \Rightarrow & e^j \end{bmatrix}$.

We give an example for case 2b which handles negation. Suppose that want to specialize the following case expression into a case expression *subproblem* which assumes that v_1 was matched against $\texttt{Cons}(x, y)$.

$$\mathit{compile}\left(\begin{array}{l}\textbf{case } v_1, v_2 \textbf{ of} \\ \begin{bmatrix} \{\} \,\&\, \neg\{\texttt{Cons}\} & \overline{D}^1_2 & \Rightarrow e^1 \\ \{\} \,\&\, \neg\{\texttt{Nil}\} & \overline{D}^2_2 & \Rightarrow e^2 \\ \textbf{default} & & \Rightarrow e_d \end{bmatrix}\end{array}\right) = \begin{array}{l}\textbf{case } v_1 \textbf{ of} \\ \begin{bmatrix} \texttt{Cons}(x, y) \Rightarrow \mathit{compile}(\mathit{subproblem}) \\ \vdots \end{bmatrix}\end{array}$$

The first clause is discarded, because the pattern $\{\} \,\&\, \neg\{\texttt{Cons}\}$ is incompatible with the assumption that v_i matches $\texttt{Cons}$. The second clause using $\{\} \,\&\, \neg\{\texttt{Nil}\}$ is compatible with the assumption. So, how can we transform this clause into a new one testing x, y and v_2? We see that pattern $\{\} \,\&\, \neg\{\texttt{Nil}\}$ does not match on the subvalues x and y of v_1, but for v_2 we still have to match against the pattern $\overline{D}^2_2$. We keep the default clause by rule 1 unchanged.

$$\mathit{subproblem} = \textbf{case } x, y, v_2 \textbf{ of} \begin{bmatrix} _ & _ & \overline{D}^2_2 & \Rightarrow e^2 \\ \textbf{default} & & & \Rightarrow e_d \end{bmatrix}$$

6.2.2 Computing Default

Next we show how to compute $\mathcal{D}(i, \{\mathcal{C}^{n_1}, \dots, \mathcal{C}^{n_z}\}, \text{Input})$ whose output reflects *the assumption that the scrutinee v_i did not match any of the head constructors $\mathcal{C}^{n_1}$ to $\mathcal{C}^{n_z}$*. This time we have no new scrutinees and only match the leftover list against a new pattern matrix $\left[D\right]$:

$$\mathcal{D}(i, \{\mathcal{C}^{n_1}, \dots, \mathcal{C}^{n_z}\}, \text{Input}) \ = \ \textbf{case } v_1, \dots, v_{i-1}, v_{i+1}, \dots, v_n \textbf{ of } \left[D\right]$$

As before, we build a new matrix $\left[D\right]$ by row-wise iteration over the clauses of the Input. A clause only contributes if its i-th pattern entry allows for a match with scrutinee v_i, knowing that it differs from all the head constructors $C^{n_1}, \dots, C^{n_z}$.

1. Row $\left[\textbf{default} \Rightarrow e_d\right]$ is added to $\left[D\right]$ unchanged.
2. Row $\left[\overline{D}^j_1 \ \dots \ \overline{D}^j_{i-1} \ \overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} \ \overline{D}^j_{i+1} \ \dots \ \overline{D}^j_n \ \Rightarrow \ e^j\right]$, only contributes according to $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}}$:
   - **a.** If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \{y_1, \dots, y_s\} \,\&\, \neg\{\mathcal{C}_1, \dots, \mathcal{C}_t\}$,
     then add the row $\left[\overline{D}^j_1 \dots \overline{D}^j_{i-1}\overline{D}^j_{i+1} \dots \ \overline{D}^j_n \quad \Rightarrow e^j[\{y_1, \dots y_s\} \mapsto v_i]\right]$.
     Note that v_i's constructor differs from all head constructors which include $\{\mathcal{C}_1, \dots, \mathcal{C}_t\}$

b. If $\overline{\mathbf{D}}^{\mathbf{j}}_{\mathbf{i}} = \|\{\overline{K}_1, \ldots, \overline{K}_t\}$, then

recursively iterate the rows of $\begin{bmatrix} \overline{D}^j_1 & \ldots & \overline{D}^j_{i-1} & \overline{\mathbf{K}}_1 & \overline{D}^j_{i+1} & \ldots & \overline{D}^j_n & \Rightarrow & e^j \\ & & & \vdots & & & & & \\ \overline{D}^j_1 & \ldots & \overline{D}^j_{i-1} & \overline{\mathbf{K}}_t & \overline{D}^j_{i+1} & \ldots & \overline{D}^j_n & \Rightarrow & e^j \end{bmatrix}$.

Let us illustrate rule 2a with the case expression below, where we perform a case split on v_1. We want to construct the default *subproblem* which assumes that v_1 did not match the constructors `Red` or `Green`.

$$compile\left(\begin{array}{l}\textbf{case } v_1, v_2 \textbf{ of} \\ \begin{bmatrix} \{\}\ \&\ \texttt{Red} & \overline{D}^1_2 & \Rightarrow e^1 \\ \{\}\ \&\ \neg\{\texttt{Green}\} & \overline{D}^2_2 & \Rightarrow e^2 \\ \textbf{default} & & \Rightarrow e_d \end{bmatrix}\end{array}\right) = \begin{array}{l}\textbf{case } v_1 \textbf{ of} \\ \begin{bmatrix} \texttt{Red} & \Rightarrow & \ldots \\ \texttt{Green} & \Rightarrow & \ldots \\ \textbf{default} & \Rightarrow & compile(subproblem) \end{bmatrix}\end{array}$$

In *subproblem*, we have no new scrutinees and only need to investigate the remaining scrutinee v_2 $subproblem = \textbf{case } v_2 \textbf{ of } [D]$. We ask for each clause if it is consistent with v_1 not looking like either `Red` or `Green`. The first clause could not match $\{\}\ \&\ \texttt{Red}$, so we discard it. For the second row, however, the test $\{\}\ \&\ \neg\{\texttt{Green}\}$ does match any v_1 that differs from both `Red` and `Green`, so we keep it. The default clause is also kept unchanged.

$$subproblem = \textbf{case } v_2 \textbf{ of } \begin{bmatrix} \overline{D}^2_2 & \Rightarrow e^2 \\ \textbf{default} & \Rightarrow e_d \end{bmatrix}$$

6.3 Compiling an Example

Let us illustrate the compilation process with a variant of the example introduced in Section 3.

$$\begin{array}{lll}\textbf{case } x \textbf{ of}\{ & y\ \&\ (\texttt{Sa} \parallel \texttt{Su}) & \Rightarrow \text{“Today is weekend!”}, \\ & y\ \&\ \neg(\texttt{Fr} \parallel \texttt{Sa} \parallel \texttt{Su}) & \Rightarrow \text{“Today is ”} +\!\!+\ \text{show}(y), \\ & \textbf{default} & \Rightarrow \text{“Tomorrow is weekend...”} \quad \} \end{array}$$

First, we rewrite this case expression into the multi-column notation, normalize all patterns (see Section 5), and abbreviate the right-hand sides by $e_1(y)$, $e_2(y)$, and d.

$$Input := \textbf{case } x \textbf{ of } \begin{bmatrix} \|\{\{y\}\ \&\ \texttt{Sa}, \{y\}\ \&\ \texttt{Su}\} & \Rightarrow & e_1(y) \\ \|\{\{y\}\ \&\ \neg\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}\} & \Rightarrow & e_2(y) \\ \textbf{default} & \Rightarrow & d \end{bmatrix}$$

We branch on the only available scrutinee x by matching it in a single-column case expression against the head constructors $\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}$. We then generate a new case expression with four clauses, one for each constructor. All that is left to do is to compute and compile the clause-specific *subproblems* of specialization and default.

$$compile\,(Input) = \textbf{case } x \textbf{ of } \begin{bmatrix} \texttt{Fr} & \Rightarrow compile\,(\mathcal{S}\,(1, \texttt{Fr}, Input)) \\ \texttt{Sa} & \Rightarrow compile\,(\mathcal{S}\,(1, \texttt{Sa}, Input)) \\ \texttt{Su} & \Rightarrow compile\,(\mathcal{S}\,(1, \texttt{Su}, Input)) \\ \textbf{default} & \Rightarrow compile\,(\mathcal{D}\,(1, \{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}, Input)) \end{bmatrix}$$

Let us consider the subproblem $\mathcal{S}\,(1, \texttt{Sa}, Input)$. We know that scrutinee x has matched `Sa`, and we have no left-over scrutinees nor new subscrutinees to consider ($n = 0$), giving $\mathcal{S}\,(1, \texttt{Sa}, Input) = \textbf{case} \quad \textbf{of}\,[S]$. We get the new clause matrix $[S]$ by iterating the old one to see how each row contributes.

- For row $\left[\|\{\{y\} \,\&\, \texttt{Sa}, \{y\} \,\&\, \texttt{Su}\} \;\Rightarrow\; e_1(y)\right]$, by rule (c), iterate $\begin{bmatrix}\{y\} \,\&\, \texttt{Sa} & \Rightarrow & e_1(y)\\ \{y\} \,\&\, \texttt{Su} & \Rightarrow & e_1(y)\end{bmatrix}$.
  - For the first one, by (a) we add the row $\left[\;\; \Rightarrow\; e_1(x)\right]$.
  - For the second one, no contribution is made.
- For row $\left[\|\{\{y\} \,\&\, \neg\{\texttt{Fr}, \texttt{Sa}, \texttt{Su}\}\} \;\Rightarrow\; e_2(y)\right]$ no contribution is made.
- For row $\left[\textbf{default} \;\Rightarrow\; d\right]$, by (1), we keep it unchanged.

This gives the specialization matrix $\left[S\right] = \begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}$.

The other specialization subproblems are computed similarly, and we obtain the following intermediate result.

$$compile\,(Input) = \textbf{case}\ x\ \textbf{of} \begin{bmatrix} \texttt{Fr} & \Rightarrow compile\left(\textbf{case}\ \ \textbf{of}\left[\textbf{default} \;\Rightarrow\; d\right]\right), \\ \texttt{Sa} & \Rightarrow compile\left(\textbf{case}\ \ \textbf{of}\begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \\ \texttt{Su} & \Rightarrow compile\left(\textbf{case}\ \ \textbf{of}\begin{bmatrix} & \Rightarrow & e_1(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \\ \textbf{default} & \Rightarrow compile\left(\textbf{case}\ \ \textbf{of}\begin{bmatrix} & \Rightarrow & e_2(x)\\ \textbf{default} & \Rightarrow & d\end{bmatrix}\right) \end{bmatrix}$$

For each of the subproblems we can apply either **Default** or **Simple** and conclude the computation.

$$compile\,(Input) = \textbf{case}\ x\ \textbf{of} \begin{bmatrix} \texttt{Fr} & \Rightarrow d \\ \texttt{Sa} & \Rightarrow e_1(x) \\ \texttt{Su} & \Rightarrow e_1(x) \\ \textbf{default} & \Rightarrow e_2(x) \end{bmatrix}$$

Note how the right-hand expression of the previous default clause d is utilized in a constructor-clause, whereas the right-hand expression of the new default-clause originates from a previously non-default clause.

6.4 Correctness of Compilation

Correctness of compilation states that the *compile* function introduced in Section 6 preserves semantics. Because the compilation algorithm operates on multi-column pattern matches, we extend wellformedness of expressions, the operational semantics and denotations to multi-column pattern matches. We can then state the correctness for the compilation algorithm:

▶ **Theorem 27.** *On a wellformed multi-column pattern match of the normalized shape Input, the compilation algorithm preserves semantics:* $[\![\, Input \,]\!] \equiv [\![\, compile(Input) \,]\!]$.

We provide the proof and the extended definitions in the appendix of the extended version of this paper [2].

7 Future Work

There are two main directions that we plan to investigate in future work.

Allowing Overlapping Patterns

In this article we enforce that no two patterns in a pattern match can overlap in order to guarantee confluence of the system, but [7, 8] showed that this restriction can be relaxed. They allow overlapping patterns under the condition that the right-hand sides of the clauses must become judgmentally equal under a substitution computed from the overlapping patterns. Let us look at an example that is allowed in their system, but not ours:

$$\begin{aligned}\mathrm{or}(x) \coloneqq \textbf{case}\ x\ \textbf{of}\ \{\ & \texttt{Pair}(\texttt{True}, _) \Rightarrow \texttt{True}, && \texttt{Pair}(_, \texttt{True}) \Rightarrow \texttt{True}, \\ & \texttt{Pair}(\texttt{False}, x) \Rightarrow x, && \texttt{Pair}(x, \texttt{False}) \Rightarrow x\ \}\end{aligned}$$

In this example, the first and the last (as well as the second and first, second and third, and the last two) patterns are overlapping. But the definition of this function is still confluent, since if we unify the overlapping patterns $\texttt{Pair}(\mathrm{True}, _)$ and $\texttt{Pair}(x, \mathrm{False})$, and apply the substitution to the right-hand sides True and x, the terms become judgmentally equal. Such overlapping patterns can be useful in a proof assistant, since they yield additional judgmental equalities that can be used to simplify a term during normalization. We think that their approach to allow overlapping patterns and our algebraic patterns can complement each other, and we therefore plan to study their interaction.

Exhaustiveness Checking

In this article, we check whether a set of patterns is overlapping, but we do not check whether a set of patterns is exhaustive. This is in spite of the fact that we are compiling these patterns using decision trees, which is a method that can easily accommodate checking the exhaustiveness of sets of patterns [22]. Checking that a given set of patterns is exhaustive is essential for real programming languages, so we plan to develop and verify an algorithm for this purpose, building on the work of [22, 15, 12]. We sketch a basic outline for such an algorithm in the appendix of the extended version of this paper [2].

8 Related Work

Order-Independent Pattern Matching

As discussed in Section 7, [7] and [8] study the issue of order-independent pattern matching in proof assistants. They want to ensure that every clause in a pattern-matching expression can be used as a judgmental equality for normalizing terms during type checking. Their system lacks advanced patterns such as or-patterns and negation patterns, but they instead relax the non-overlap requirement.

Substructural logics

And-patterns and wildcard patterns are strongly related to the rules of contraction and weakening in the sequent calculus. This relationship between structural rules and certain forms of patterns has been observed by [4], who present a pattern matching calculus for the sequent calculus which includes both and-patterns and wildcard-patterns. This relationship can also be observed in programming languages with substructural type systems, such as Rust, where the expression `match v {x@y => (x, y)}` does not type check if the type of v does not support contraction, i.e. does not implement the `Copy` trait.

Logic Programming and Datalog

The problems with the binding structure of patterns involving negations that we discussed in this paper appears in a similar form for Datalog programs. In this article we are only interested in the binding occurrences of variables in a pattern, i.e. those variables which occur under an even number of negations, and which can be used in the right-hand side of a clause. Occurrences of logic variables in a Datalog program can be distinguished into binding occurrences and bound occurrences, where binding occurrences assign a value to a logic variable and bound occurrences can access the value the variable is bound to. The authors of [18] describe a type system which tracks binding and bound occurences of variables and describe the rules for disjunctions and negations in their Section 4.3. Their rule for negations is:

$$\frac{\Gamma_1 \vdash^{\# \cdot -} a \; \texttt{ok} \dashv \Gamma_2}{\Gamma_1 \vdash^{\#} \texttt{not} \; a \; \texttt{ok} \dashv \Gamma_2} \text{ A-Not}$$

Here # stands for a polarity, either + or −, which controls which variables in the input context Γ_1 are already bound, and the multiplication with − correspondingly switches this interpretation. This rule therefore operates very similarly to our pattern typing rule P-Neg in Section 4.3 and guarantees that negation is involutive with respect to the typing relation.

Dynamic First Class Patterns

We use "dynamic first class patterns" for systems in which patterns can be computed and passed at runtime. Such systems, whose theory was studied by [14] and [13], can also express our algebraic patterns. The technical report [11] describes an extension of the object-oriented language Newspeak (cf. [3]) by such a system of dynamic pattern matching. They explicitly discuss various *pattern combinators*, i.e. combinators which combine patterns to yield new ones. They discuss disjunction, conjunction and negation pattern combinators which correspond to our or-patterns, and-patterns and negation patterns. In this article, we focus exclusively on statically-known patterns, since these can be analyzed and compiled more easily.

Negation Patterns

Negation patterns also appear in [19], where the author mentions that "Having false- and or-patterns allows us to define the complement of a pattern – for any pattern at a type A, we can define another pattern that matches exactly the values of type A the original does not". [19] presents a translation that eliminates negation patterns, but this translation makes a closed-world assumption about all the available constructors for each data type. For example, he translates the negated pair pattern $\neg\texttt{Pair}(p_1, p_2)$ by the pattern $\texttt{Pair}(\neg p_1, _) \parallel \texttt{Pair}(_, \neg p_2)$, whereas we (in Section 5.1) translate it by the pattern $\neg\texttt{Pair}(_, _) \parallel \texttt{Pair}(\neg p_1, _) \parallel \texttt{Pair}(_, \neg p_2)$. He can omit the first disjunct of the or-pattern because he knows that the pair type only has one constructor. If we extend his translation scheme to the list type, then we would translate the negation of the constructor-pattern Nil to the pattern Cons(_, _), since we know that the list type only has two constructors. Because we encode negative information directly in patterns, we never have to rely on type information during our compilation process. Negation patterns have also been introduced under the name of "anti-patterns" and "anti-pattern matching" by [16, 17, 6]. The formalism they present differs from ours in the following points: They only allow constructor-patterns, variable patterns and negation patterns, which means that they cannot give a recursive

translation for patterns involving negations, since that also requires or-patterns. Their formalism is also more rooted in the theory of term rewriting systems instead of programming languages, and they do not present a compilation algorithm to simple patterns as we do.

Pattern matching compilation algorithms

A simple way to execute pattern matching expressions with nested patterns is to try one clause after another, until the first one matches. This, however, is extremely inefficient if we have multiple clauses which are headed by the same outer constructor. In that case, we are performing the same tests multiple times. As an introduction to this problem, we can still recommend the book by [24, Chapter 5]. There are many algorithms that compile nested pattern matches to efficient code, but most of them fall into one of two categories. Nested pattern matching can be either compiled using backtracking automata [1, 20] or by compiling to decision trees [23]. The main difference between these two approaches is a classical time vs. space trade-off. Compiling to decision trees guarantees that every test is executed at most once, but can potentially increase the size of the generated code; backtracking automata do not increase the size of the code, but may perform some tests multiple times. In Section 6 we present a version of the algorithm by [23] which compiles to decision trees. Recently, Cheng and Parreaux [5] also proved the correctness of a pattern matching compilation algorithm for their very expressive pattern matching syntax; they, however, do not care about order-independence in the same way that we do.

9 Conclusion

Pattern matching is an extremely popular declarative programming construct. We argued that if we want to make pattern matching even more declarative, then the order of the clauses in a program should not matter. But if we require that two patterns must not overlap, and if we want to avoid overly verbose pattern matching expressions, then we have to make the language of patterns and pattern matching more expressive. We introduced two complementary features, a boolean algebra of patterns and default clauses, which together solve the verbosity problem. We have provided the operational and static semantics of these constructs, and have shown that they can be compiled to efficient code.

Data-Availability Statement

The theorems in Section 3 have been formalized and checked in the proof assistant Rocq. These proofs are available as related material.

References

1 Lennart Augustsson. Compiling pattern matching. In Jean-Pierre Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 368–381, Berlin, Heidelberg, 1985. Springer. doi:10.1007/3-540-15975-4_48.
2 David Binder and Lean Ermantraut. The algebra of patterns (extended version). *arXiv*, 2025.
3 Gilad Bracha. Newspeak programming language draft specification version 0.104, 2022. Technical report.
4 Serenella Cerrito and Delia Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1):71–127, 2004. doi:10.1016/j.tcs.2004.03.032.
5 Luyu Cheng and Lionel Parreaux. The ultimate conditional syntax. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689746.

6 Horatiu Cirstea, Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-patterns for rule-based languages. *Journal of Symbolic Computation*, 45(5):523–550, 2010. Symbolic Computation in Software Science. doi:10.1016/j.jsc.2010.01.007.

7 Jesper Cockx. Overlapping and order-independent patterns in type theory. Master's thesis, KU Leuven, 2013. URL: https://lirias.kuleuven.be/retrieve/262314/.

8 Jesper Cockx, Frank Piessens, and Dominique Devriese. Overlapping and order-independent patterns. In Zhong Shao, editor, *Programming Languages and Systems*, pages 87–106, Berlin, Heidelberg, 2014. Springer. doi:10.1007/978-3-642-54833-8_6.

9 Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 103(2):235–271, 1992. doi:10.1016/0304-3975(92)90014-7.

10 Jacques Garrigue. Programming with polymorphic variants. In *ML workshop*. Baltimore, 1998. URL: https://caml.inria.fr/pub/papers/garrigue-polymorphic_variants-ml98.pdf.

11 Felix Geller, Robert Hirschfeld, and Gilad Bracha. *Pattern Matching for an Object-Oriented and Dynamically Typed Programming Language*. Technische Berichte des Hasso-Plattner-Instituts für Softwaresystemtechnik an der Universität Potsdam. Universitätsverlag Potsdam, 2010.

12 Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. Lower your guards: A compositional pattern-match coverage checker. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020. doi:10.1145/3408989.

13 Barry Jay. *Pattern Calculus: Computing with Functions and Structures*. Springer, 2009. doi:10.1007/978-3-540-89185-7.

14 Barry Jay and Delia Kesner. First-class patterns. *Journal of Functional Programming*, 19(2):191–225, 2009. doi:10.1017/S0956796808007144.

15 Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. Gadts meet their match: Pattern-matching warnings that account for gadts, guards, and laziness. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 424–436, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2784731.2784748.

16 Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching. In Rocco De Nicola, editor, *Programming Languages and Systems*, pages 110–124, Berlin, Heidelberg, 2007. Springer. doi:10.1007/978-3-540-71316-6_9.

17 Claude Kirchner, Radu Kopetz, and Pierre-Etienne Moreau. Anti-pattern matching modulo. In Carlos Martín-Vide, Friedrich Otto, and Henning Fernau, editors, *Language and Automata Theory and Applications*, pages 275–286, Berlin, Heidelberg, 2008. Springer. doi:10.1007/978-3-540-88282-4_26.

18 David Klopp, Sebastian Erdweg, and André Pacak. A typed multi-level datalog ir and its compiler framework. *Proc. ACM Program. Lang.*, 8(OOPSLA2), October 2024. doi:10.1145/3689767.

19 Neelakantan R. Krishnaswami. Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, 2009. Association for Computing Machinery. doi:10.1145/1480881.1480927.

20 Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, ICFP '01, pages 26–37, New York, NY, USA, 2001. Association for Computing Machinery. doi:10.1145/507635.507641.

21 Justin Lubin and Sarah E. Chasins. How statically-typed functional programmers write code. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485532.

22 Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17(3):387–421, 2007. doi:10.1017/S0956796807006223.

23 Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, 2008. `doi:10.1145/1411304.1411311`.

24 Simon Loftus Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall International Series in Computer Science. Prentice Hall, 1987.

25 Nick Rioux, Xuejing Huang, Bruno C. d. S. Oliveira, and Steve Zdancewic. A bowtie for a beast: Overloading, eta expansion, and extensible data types in f. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. `doi:10.1145/3571211`.

26 Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter. *University of Virginia, Charlottesville, VA*, 2000. URL: `https://www.cs.tufts.edu/~nr/cs257/archive/norman-ramsey/match.pdf`.

27 Titus Winters, Tom Manshreck, and Hyrum Wright. *Software Engineering at Google – Lessons Learned from Programming Over Time.* O'Reilly Media, Inc., 2020.

28 Weixin Zhang and Bruno C. d. S. Oliveira. Pattern matching in an open world. *SIGPLAN Not.*, 53(9):134–146, April 2020. `doi:10.1145/3393934.3278124`.

29 Weixin Zhang, Yaozhu Sun, and Bruno C. D. S. Oliveira. Compositional programming. *ACM Trans. Program. Lang. Syst.*, 43(3), September 2021. `doi:10.1145/3460228`.