Coverage Semantics for Dependent Pattern Matching

Under Review for ESOP 2025

JOSEPH EREMONDI, University of Regina, Canada
OHAD KAMMAR, University of Edinburgh, Scotland, United Kingdom

Dependent pattern matching is a key feature in dependently typed programming. However, there is a theory-practice disconnect: while many proof assistants implement pattern matching as primitive, theoretical presentations give semantics to pattern matching by elaborating to eliminators. Though theoretically convenient, eliminators can be awkward and verbose, particularly when for complex combinations of patterns.

This work aims to bridge the theory-practice gap by presenting a direct categorical semantics for pattern matching, which does not elaborate to eliminators. This is achieved using sheaf theory to describe when sets of arrows (terms) can be amalgamated into a single arrow. We present a language with top-level dependent pattern matching, without specifying which sets of patterns are considered covering for a match. Then, we give a sufficient criterion for which pattern-sets admit a sound model: patterns should be in the canonical coverage for the category of contexts. Finally, we use sheaf-theoretic saturation conditions to devise some allowable sets of patterns. We are able to express and exceed the status quo, giving semantics for datatype constructors, nested patterns, absurd patterns, propositional equality, and dot patterns.

Additional Key Words and Phrases: categorical semantics, dependent pattern matching, sheaf theory, coverages

1 INTRODUCTION

Pattern matching is a core feature in dependently typed programming. With pattern matching one can specify a function consuming an input by giving functions for every possible way that input might have been constructed. For dependent types, the defined function can be a universally quantified proof, giving a Curry-Howard analogue of proof-by-cases.

However, there is a disconnect between the theory and practice of dependent pattern matching. Many dependently typed languages take pattern matching as a built-in user-facing construct: Coq [Bertot and Castéran 2004], Agda [Norell 2009], and Idris [Brady 2021] all contain a form of dependent pattern matching in their core calculi. However, most theoretical treatments of dependent types deal with *eliminators* [McBride 2002]: primitive functions for an inductive type that take an instance of the target type for each return type. Eliminators and pattern matching are equally expressive, both with Axiom K [Goguen et al. 2006] and without [Cockx et al. 2014a].

While it is possible to express all pattern matches using eliminators, it is not always convenient. Some pattern matching features require lengthy translations when converting to eliminators, such as overlapping patterns, catch-all branches, or matching on multiple values at once. Moreover, languages differ in which pattern matches they allow, so every variant of pattern matching requires a new eliminator-translation to prove consistency. Even the implementations of dependently typed languages are restricted by eliminators, since most pattern matches are elaborated into case trees with a 1:1 correspondence between branches and constructors of an inductive type.

The contribution of this paper is to narrow the theory-practice divide with a highly general syntax (Section 2) and categorical semantics for dependent pattern matching (Section 3). The semantics is direct and generic: pattern matches are translated directly into semantic objects without desugaring to eliminators or case trees, and the semantics is parameterized over an abstract coverage specifying which sets of patterns one can match against. We investigate the vision set forth by

Authors' addresses: Joseph Eremondi, Department of Computer Science, University of Regina, Canada, jeremondi@uregina.ca; Ohad Kammar, Laboratory for the Foundation of Computer Science, University of Edinburgh, Scotland, United Kingdom.

Epigram [McBride 2005; Mcbride and Mckinna 2004] to enable diverse pattern-matching abstractions that go beyond the list of constructors declared by a datatype. We focus on non-overlapping patterns, but describe a potential road map to supporting pattern overlap.

In constructing our generic semantics, we present a general sufficient criterion for when a coverage leads to well-defined pattern matches without compromising logical consistency (Section 4), mechanized in Lean. We define this criterion, drawing on parallels between dependent pattern matching and the theory of sheaves on a site, discovering that it is sufficient for each allowed set of patterns to correspond to a cover in the canonical coverage for the semantic category. Moreover, we use elementary results from sheaf theory to describe a group of closure operations which preserve the canonicity of a coverage (Section 5). These give a simple and direct way to model common features like multi-value matches, nested patterns, and matching on propositional equality proofs. With the exception of recursion, we achieve feature parity with the original presentation of dependent pattern matching by Coquand [1992]. We conclude with an illustrative example (Section 6) and related and future work (Section 7).

A major contribution of our work is expressing pattern matching in the language of categories and sheaves. Our approach is semantic: instead of elaborating pattern matching syntax, we take pattern matches as primitive and work directly in the semantic domain, avoiding the need to consider eliminators as a syntactic primitive. The connection has been implicit for many decades, but we make it formal. That said, we only assume basic knowledge of functors, pullbacks, and slice categories, and we present all the required sheaf theory.

2 COVERTT: THE SOURCE LANGUAGE

We begin with a variant of Martin Löf Type Theory, called CoverTT, parameterized over which sets of patterns can be matched against. Drawing from sheaf-theory terminology, when a set of patterns is permissible on the left-hand side of a pattern match we call it a **cover**, and say it is **covering**. The set of all covers together is called a **coverage**. The main distinct feature of CoverTT is that it is parameterized over a coverage.

2.1 The Anatomy of a Datatype

As they are presented in most dependently typed languages, inductively defined datatypes conflate four different concepts. Consider the quintessential inductive family of length-indexed vectors:

$$\begin{aligned}&\texttt{data}\ \mathsf{Vec}\ (A : \mathcal{U}) : (m : \mathbb{N}) \to \mathcal{U}\ \texttt{where}\\ &\qquad \mathsf{nil} : \mathsf{Vec}\ A\ 0\\ &\qquad \mathsf{cons} : (n : \mathbb{N}) \to A \to \mathsf{Vec}\ A\ n \to \mathsf{Vec}\ A\ (n+1)\end{aligned}$$

This definition implicitly relies on the following concepts, which we design CoverTT to separate.

- **Coproducts:** An inductive type behaves like the sum of its constructor types. For Vec, it behaves like $\mathbb{1} + (A \times \mathsf{Vec}\ A\ n)$. In CoverTT, each inductive I is declared at the top level to have a finite collection of constructors $D^I_1 \dots D^I_n$, each of which has a type ending in I.
- **Dependent fields:** each constructor has a dependent product type, so the arguments to a constructor are a curried dependent record, where the types of later fields can depend on the values of earlier ones. For Vec, in cons the return type and the type of the tail of the list depend on the earlier parameter n. In CoverTT, the type of a constructor is given as a telescope, where the types of later entries are allowed to depend on the values of previous entries.
- **Indexing:** each constructor implies a specific equation about the index values. For Vec, nil restricts that $m = 0$ and cons restricts that $m = n+1$. In CoverTT, we use **Fording** [McBride

2000], where each constructor has an identical return type, but may constrain type parameters using equality-proof fields. We treat equality as primitive and make it the only way to constrain indices of a constructor, simplifying the models of CoverTT.
- **Recursion:** inductive types can refer to themselves in fields, except to the left of a function arrow, a condition known as strict positivity. Typically, one can define structurally recursive functions over Vec. Here, Vec occurs as a field type for cons, which is allowed because it is not to the left of an arrow type. We omit recursion from CoverTT, as we believe it requires a separate toolkit of abstractions. In Section 7.2.3 we discuss its possible addition. In examples, we refer to some inductive types defined using self-reference, such as vectors or natural numbers.

An example of how these concepts appear in CoverTT can be seen in Section 2.4.

Restricting to Top-Level Datatypes. CoverTT only allows data types and pattern matches to be declared at the top level. The parameter and constructor types of datatypes must be typeable in the empty context, though they can refer to other data types. Likewise, the branches and motive of a pattern match must be typeable in the empty context. These assumptions simplify the presentation of CoverTT while reflecting how datatypes are implemented in languages like Agda, where declarations in a non-empty context are desugared into top level declarations with extra parameters. Treating nested patterns and arbitrary coverages is an interesting problem even with top-level matches, and requires substantial technical developments even to handle ordinary matching.

2.2 Syntax and Typing

The syntax for CoverTT below is standard, except for the separation-of-concerns from Section 2.1 Figure 1 gives the typing rules for CoverTT.

$$\begin{array}{rcl} \textsc{Term} \ni s, S, t, T & ::= & x \mid \mathcal{U} \mid \Pi(x : S).\, T \mid \lambda x.\, t \mid t\ s \mid s =_T t \mid \mathtt{refl}_t \\ & & \mid I\ \mathbf{t} \mid D^I\ \mathbf{t} \mid \mathtt{case}\ (\mathbf{t} : \Gamma)\ \mathtt{to}\ T\ \mathtt{of}\ \{\overrightarrow{\Delta_i.\, \mathbf{s}_i \Rightarrow t}^{i}\} \\ \textsc{Ctx} \ni \Gamma, \Delta, \Xi & ::= & \cdot \mid \Gamma, (x : T) \\ \textsc{Subst} \ni \mathbf{s}, \mathbf{t} & ::= & \cdot \mid \mathbf{s} \mathbin{;} t \end{array}$$

Overline arrows denote sequences, while bold metavariables denote dependent sequences, e.g. substitutions. Variables are assigned types from the context. Typing for dependent functions and equality is standard, with rules for their types, introduction, and elimination. The equality type is intensional, and there is no reflection rule by which propositional equalities can be made judgmental equalities. Nevertheless, CoverTT is consistent with extensional models that identify all propositionally-equal terms, and we will primarily be concerned with such models. We have no J-axiom, instead using the coverage to specify how to match on $\mathtt{refl}_t$. We have one universe which does not have a type, but we believe a universe hierarchy is compatible with our semantics, so we leave it for future work.

2.2.1 Contexts and Substitutions. Figure 2 also specifies well-formedness rules for contexts and substitutions. Contexts are sequences of typed variables, where later types may refer to variables from earlier in the context. Substitutions are the inhabitants of contexts. The rules are like an iterated version of dependent pairs: the type of the later values in the substitutions may depend on the earlier values. We borrow the notation $\Gamma \vdash \mathbf{t} \Rightarrow \Delta$ from Hofmann [1997], since such a substitution corresponds to a morphism $\Gamma \longrightarrow \Delta$ in the models we define. We use the notation $[\mathbf{s}/\Delta]t$ to denote the simultaneous substitution of the variables bound in Δ by the terms of $\mathbf{s}$ in t. If $\Delta \vdash t : T$, then $\Gamma \vdash [\mathbf{s}/\Delta]t : [\mathbf{s}/\Delta]T$.

$$\boxed{\Gamma \vdash t : T \quad \textit{(Well Typed Terms and Types)}} \qquad \boxed{\{t_i\}_i \rhd T \quad \textit{(Covering Patterns, generic relation)}}$$

$$\textsc{TypeConv}\ \frac{\begin{array}{c}\Gamma \vdash t : S \\ \Gamma \vdash S \equiv T : \mathcal{U}\end{array}}{\Gamma \vdash t : T} \qquad \textsc{TyVar}\ \frac{\begin{array}{c}\vdash \Gamma\ \mathsf{ctx} \\ (x : T) \in \Gamma\end{array}}{\Gamma \vdash x : T} \qquad \textsc{TyPi}\ \frac{\begin{array}{c}\Gamma \vdash S : \mathcal{U} \\ \Gamma, (x : S) \vdash T : \mathcal{U}\end{array}}{\Gamma \vdash \Pi(x : S).\, T : \mathcal{U}} \qquad \textsc{TyApp}\ \frac{\begin{array}{c}\Gamma \vdash t : \Pi(x : S).\, T \\ \Gamma \vdash s : S\end{array}}{\Gamma \vdash t\ s : T[s/x]}$$

$$\textsc{TyLam}\ \frac{\Gamma, (x : S) \vdash t : T}{\Gamma \vdash \lambda x.\, t : \Pi(x : S).\, T} \qquad \textsc{TyEq}\ \frac{\Gamma \vdash T : \mathcal{U} \qquad \Gamma \vdash s : T \qquad \Gamma \vdash t : T}{\Gamma \vdash s =_T t : \mathcal{U}} \qquad \textsc{TyRefl}\ \frac{\Gamma \vdash t : T}{\Gamma \vdash \mathtt{refl}_t : t =_T t}$$

$$\textsc{TyInd}\ \frac{\mathsf{Params}(I) := \Delta \qquad \Gamma \vdash \mathbf{t} \Rightarrow \Delta}{\Gamma \vdash I\ \mathbf{t} : \mathcal{U}} \qquad \textsc{TyCtor}\ \frac{\mathsf{Fields}(D^I) := \Delta, \Xi \qquad \Gamma \vdash \mathbf{s} \Rightarrow \Delta \qquad \Gamma \vdash \mathbf{t} \Rightarrow [\mathbf{s}/\Delta]\Xi}{\Gamma \vdash D^I\ \mathbf{t} : I\ \mathbf{s}}$$

$$\textsc{TyCase}\ \frac{\begin{array}{c}\vdash \Xi\ \mathsf{ctx} \qquad \Gamma \vdash \mathbf{t}_{\mathrm{scrut}} \Rightarrow \Xi \qquad \Xi \vdash T_{motive} : \mathcal{U} \qquad \overrightarrow{\vdash \Delta_i\ \mathsf{ctx}}^i \qquad \overrightarrow{\Delta_i \vdash \mathbf{s}_i \Rightarrow \Xi}^i \\ \vdash \{\mathbf{s}_i\}_i \rhd \Xi \qquad \overrightarrow{\Delta_i \vdash t_i : [\mathbf{s}_i/\Xi]T_{motive}}^i\end{array}}{\Gamma \vdash \mathsf{case}\ (\mathbf{t}_{\mathrm{scrut}} : \Xi)\ \mathsf{to}\ T_{motive}\ \mathsf{of}\ \{\overrightarrow{\Delta_i.\, \mathbf{s}_i \Rightarrow t_i}^i\} : [\mathbf{t}_{\mathrm{scrut}}/\Xi]T_{motive}}$$

Fig. 1. CoverTT: Term Typing

$$\boxed{\vdash \Gamma\ \mathsf{ctx} \quad \textit{(Well Formed Contexts)}} \qquad \boxed{\Gamma \vdash \mathbf{t} \Rightarrow \Delta \quad \textit{(Well Formed Substitutions)}}$$

$$\textsc{CtxNil}\ \frac{}{\vdash \cdot\ \mathsf{ctx}} \qquad \textsc{CtxCons}\ \frac{\vdash \Gamma\ \mathsf{ctx} \qquad \Gamma \vdash T : \mathcal{U}}{\vdash \Gamma, (x : T)\ \mathsf{ctx}} \qquad \textsc{EnvNil}\ \frac{}{\Gamma \vdash \cdot \Rightarrow \cdot} \qquad \textsc{EnvCons}\ \frac{\Gamma \vdash \mathbf{s} \Rightarrow \Delta \qquad \Gamma \vdash t : [\mathbf{s}/\Delta]T}{\Gamma \vdash \mathbf{s} ⨟ t \Rightarrow \Delta, (x : T)}$$

Fig. 2. Typing: Contexts and Substitutions

2.2.2 Datatype and Pattern Matching Syntax. We assume a fixed collection of inductive type constructors, along with data constructors. For each datatype there is a fixed context of parameters $\mathsf{Params}(I)$ and, for each constructor, fields $\mathsf{Fields}(D^I)$, such that $\vdash \mathsf{Params}(I), \mathsf{Fields}(D^I)$, i.e., both are well-formed and the fields may depend on the parameters. The type $I\ \mathbf{t}$ denotes the type constructor I applied to parameters $\mathbf{t}$. $D^I\ \mathbf{t}$ is the data constructor D for the type I, given $\mathbf{t}$ for its fields. We omit I in D^I when it is clear from context.

The pattern matching form is a nameless version of defining functions by multiple pattern matching clauses, as in Agda or Idris. The term

$$\mathsf{case}\ (\mathbf{t} : \Xi)\ \mathsf{to}\ T\ \mathsf{of}\ \{\overrightarrow{\Delta_i.\, \mathbf{s}_i \Rightarrow t_i}^i\}$$

denotes a match on **scrutinees** $\mathbf{t}$ of context-type Ξ, producing a result of the **motive** type T, where T may refer to the variables bound in Ξ. The branches, indexed by i, each have a left-hand-side pattern $\mathbf{s}_i$, which may contain pattern variables from the context Δ_i, and which produces a result t_i. The scrutinee is a substitution, rather than a term, because pattern matching functions can take multiple arguments, and the types of later arguments can depend on the values of earlier ones.

2.2.3 Typing Pattern Matches. The TyCase typing rule (Fig. 1) is the most important rule. To type $\Gamma \vdash \texttt{case}\ (\mathbf{t_{scrut}} : \Xi)\ \texttt{to}\ T_{motive}\ \texttt{of}\ \{\overrightarrow{\Delta_i.\,\mathbf{s}_i \Rightarrow t_i}^{i}\}$, the scrutinee inhabits scrutinee context. Because pattern matching is dependent, the motive T_{motive} is indexed over the scrutinee type. The pattern $\mathbf{s_i}$ for each branch must inhabit the scrutinee type Ξ in the context of its pattern variables Δ_i. Each branch is typed against the context of its pattern variables. A branch must inhabit the motive for the scrutinee value given by that branch's pattern, e.g., $\mathbf{s_i}$. Finally, the entire match inhabits the motive type, instantiated to the scrutinee.

Critically, the motive T_{motive} and each branch t_i must be typeable in the closed context Δ_i, making no reference to Γ. This restriction matches practice: in Agda and Idris, pattern matching definitions elaborate to top level declarations.

2.2.4 The Generic Coverage Relation. In a pattern match, the patterns $\overline{\mathbf{s_i}}$ must be covering. At no point do we require that the left-hand side of a pattern match correspond to constructors of an inductive type, or even that the matched upon type be an inductive type. Instead, we appeal to an arbitrary judgment $\{\mathbf{s_i}\}_i \triangleright \Xi$, which is to be read as "the patterns $\mathbf{s_1} \dots \mathbf{s_n}$ are a total decomposition of the context Ξ". This replaces the usual condition that there must be a case for each constructor of the datatype. This relation is the parameter by which we can tune CoverTT, and it can have a variety of forms, from requiring a single scrutinee with exactly one branch per constructor, to allowing multiple scrutinees with arbitrary nested patterns and absurd branches omitted. In Section 4.2, we explore the conditions that a coverage must satisfy to result in a well-behaved type theory. In general, we expect that a coverage will consist of a basis set of coverings, containing at least variables and constructors for each datatype. This set can then be closed under composition, concatenation, etc. To maintain generality we do not require closure conditions, but in Section 5 we justify why we can always impose them.

2.3 Computational Rules

Figure 3 defines definitional equality for CoverTT. We omit structural rules (reflexivity, symmetry, transitivity) and congruence rules for brevity. Rule EqApp is the usual β-reduction for functions.

In EqMatch, we reduce a pattern match when the scrutinee is equal to $[\mathbf{t_{match}}/\Delta_j]\mathbf{s}_j$ for some $\Gamma \vdash \mathbf{t_{match}} \Rightarrow \Delta_j$, i.e., it is a pattern applied to some value for each pattern variable of $\mathbf{s_j}$. This substitution $\mathbf{t_{match}}$ instantiates the pattern variables in the right-hand side t_j, which becomes the value of the entire match.

2.4 Example: Vectors

To see these constructs concretely, we show how length-indexed vectors from Section 2.1 would be represented in our system, along with a type safe head function. In CoverTT, vectors can be defined using labelled sums. Notice the equality-type fields which encode the index constraints via Fording.

$$\mathsf{Params}(\mathsf{Vec}) := \cdot, (A : \mathcal{U}), (n : \mathbb{N})$$
$$\mathsf{Fields}(\mathsf{Nil}^{\mathsf{Vec}}) := \cdot, (A : \mathcal{U}), (n : \mathbb{N}), (eq : n = 0)$$
$$\mathsf{Fields}(\mathsf{Cons}^{\mathsf{Vec}}) := \cdot, (A : \mathcal{U}), (n : \mathbb{N}), (m : \mathbb{N}), (h : A), (t : \mathsf{Vec}\ A\ m), (eq : n = m + 1)$$

In Agda-style notation, a type safe head function for vectors looks like the following:

$$\mathsf{head} : (A : \mathcal{U}) \to (n : \mathbb{N}) \to \mathsf{Vec}\ A\ (1 + n) \to A$$
$$\mathsf{head}\ A\ n\ (\mathsf{Cons}\ h\ t) = h$$

$$\boxed{\Gamma \vdash t = s : T \ \textit{(Term Definitional Equality)}} \ \boxed{\vdash \Gamma \equiv \Delta \ \mathsf{ctx} \ \textit{(Equal Contexts)}}$$
$$\boxed{\Gamma \vdash \mathbf{t} \Rightarrow \Delta \ \textit{(Equal Substitutions)}}$$

EqMatch
$$\frac{\begin{array}{ccccc} & \vdash \Xi\ \mathsf{ctx} & \overrightarrow{\vdash \Delta_i\ \mathsf{ctx}}^{i} & j \in \{1 \ldots i\} & \\ \Gamma \vdash \mathbf{t}_{\mathbf{match}} \Rightarrow \Delta_j & \Xi \vdash T : \mathcal{U} & \overrightarrow{\Delta_i \vdash \mathbf{s}_i \Rightarrow \Xi}^{i} & \vdash \{\mathbf{s}_i\}_i \gtrdot \Xi & \overrightarrow{\Delta_i \vdash t_i : [\mathbf{s}_i/\Xi]T}^{i} \end{array}}{\Gamma \vdash \mathtt{case}\ ([\mathbf{t}_{\mathbf{match}}/\Delta_i]\mathbf{s}_j : \Xi)\ \mathtt{to}\ T\ \mathtt{of}\ \{\overrightarrow{\Delta.\, \mathbf{s}_i \Rightarrow t_i}^{i}\} \equiv [\mathbf{t}_{\mathbf{match}}/\Delta_j]t_j : [\mathbf{t}_{\mathbf{match}}/\Delta_j][\mathbf{s}_j/\Xi]T}$$

EqApp
$$\frac{\Gamma, (x : S) \vdash t : \Pi(x : S).\, T \qquad \Gamma \vdash s : S}{\Gamma \vdash (\lambda x.\, t)\ s \equiv [s/x]t : [s/x]T}$$

EqCtxNil
$$\frac{}{\vdash \cdot \equiv \cdot\ \mathsf{ctx}}$$

SubNil
$$\frac{}{\Gamma \vdash \cdot \equiv \cdot : \cdot}$$

EqCtxCons
$$\frac{\vdash \Gamma \equiv \Gamma'\ \mathsf{ctx} \qquad \Gamma \vdash T \equiv T' : \mathcal{U}}{\vdash \Gamma, (x : T) \equiv \Gamma', (x : T')\ \mathsf{ctx}}$$

SubCons
$$\frac{\Gamma \vdash \mathbf{s} \equiv \mathbf{s}' : \Delta \qquad \Gamma \vdash t \equiv t' : [\mathbf{s}/\Delta]T}{\Gamma \vdash \mathbf{s} \fatsemi t \equiv \mathbf{s}' \fatsemi t' : \Delta, (x : T)}$$

Fig. 3. Definitional Equality: Computational Rules

In CoverTT this would be defined as:

$$\mathsf{head} : \Pi(A : \mathcal{U})(n : \mathbb{N})(x : \mathsf{Vec}\ A\ (n+1)).\, A$$
$$\mathsf{head} := \lambda A.\, \lambda n.\, \lambda x.\, \mathtt{case}\ (\cdot \fatsemi n \fatsemi x : \cdot, (n : \mathbb{N}), (x : \mathsf{Vec}\ A\ (n+1)))\ \mathtt{to}\ A\ \mathtt{of}\ \{$$
$$\quad \cdot, (m : \mathbb{N}), (h : A), (t : \mathsf{Vec}\ A\ m).\, \cdot \fatsemi m \fatsemi \mathsf{Cons}\ A\ (m+1)\ m\ h\ t\ \mathtt{refl}_{m+1} \Rightarrow h\}$$

That is, the function takes A, n, and x as type, number, and vector parameters. It then passes those parameters as the scrutinee of the pattern match, which is annotated with their types. The annotation is a telescope of types, so it introduces new names for the scrutinees. It happens that here we are passing names as the scrutinees, but this need not be the case, which is why we need new names for them. The result type is annotated as A. The match has a single branch, with a telescope of pattern variables m, h and t with their types. The pattern for the branch has $m + 1$ as the value for scrutinee n and Cons applied to its arguments for scrutinee x. Finally, to the right of $\Rightarrow$ is the result for this case, which is h.

Implicit in this example is the need for the following to hold:

$$\{\cdot, (m : \mathbb{N})(h : A), (t : \mathsf{Vec}\ A\ m).\, \cdot \fatsemi m \fatsemi \mathsf{Cons}\ A\ (m+1)\ m\ h\ t\ \mathtt{refl}_{m+1}\} \gtrdot \cdot, (n : \mathbb{N}), (x : \mathsf{Vec}\, A\ (n{+}1))$$

That is, the pattern for the single branch needs to cover the scrutinee type. Deducing that these patterns are a valid coverage involves seeing that $\mathtt{refl}_{m+1}$ constrains the value of n and that Nil has an absurd type.

The goal of this paper is to define direct semantics that justify such deductions, and give a broad framework to define valid coverages.

3 CATEGORICAL MODELS OF COVERTT

In this section, we translate the syntactic constructs of CoverTT into the language of **Categories with Families** (CwFs). CwFs correspond almost exactly to the syntactic structure of dependent type theory, but with syntactic substitution replaced by a semantic operation, and with implicit liftings between contexts made explicit.

3.1 Background: Categories with Families

Here we recapitulate the definition and notation of CwFs, a categorical model for dependent type theory that follows the syntax fairly closely. See Hofmann [1997] for more details.

Recall that a **family** X is a pair $(I_X, \underline{X})$ consisting of a set I_X and an I-indexed sequence of sets $(\underline{X}_i)_{i \in I}$. A **map** of families $f : X \to Y$ is a pair $(f_I, \underline{f})$ consisting of a function $f_I : I_X \to I_Y$ and an I-indexed sequence of functions $(f_i : \underline{X}_i \to \underline{Y}_{fi})_{i \in I}$. The category **Fam** has families as objects and maps of families as morphisms, with the componentwise identity and composition structure.

A **basic CwF** C is a pair (C_o, F) consisting of a category C_o and a functor $F : C_o^{op} \to$ **Fam**. The functor F packs four pieces of structure which we'll unpack using the following notation:

- We call the objects $\Gamma \in C_o$ **contexts**, and the morphisms $\theta : \Gamma \to \Delta$ **substitutions**.
- For every context $\Gamma \in C_o$, we denote the family $F\Gamma$ by $(\mathsf{Ty}(\Gamma), \mathsf{Tm}_\Gamma(\_))$. We call the elements of its indexing set $\mathsf{Ty}(\Gamma)$ the **types** in context Γ. For each type in $T \in \mathsf{Ty}(\Gamma)$, we call the element of the component $\mathsf{Tm}_\Gamma(T)$ the **terms** of type T in context Γ. We omit the subscript Γ when it is clear from context.
- For every substitution $\theta : \Gamma \to \Delta$, we have a map of families $F\theta : F\Gamma \leftarrow F\Delta$, and we use the same notation for both its components $F\theta = (\_\{\theta\}, \_\{\theta\})$ and call both **substitution functions**. The first component is the substitution function on **types** $\_\{\theta\} : \mathsf{Ty}(\Gamma) \leftarrow \mathsf{Ty}(\Delta)$. The second component is a sequence of substitution functions on the terms of each type $\_\{\theta\} : \mathsf{Tm}_\Gamma(T\{\theta\}) \leftarrow \mathsf{Tm}_\Delta(T)$.
- The functoriality of F amounts to the following four properties, which we call the **substitution lemma** for this CwF, where T ranges over $\mathsf{Ty}(\Gamma)$, t ranges over $\mathsf{Tm}_\Gamma(T)$:

$$T\{\mathrm{id}_\Gamma\} = T \quad t\{\mathrm{id}_\Gamma\} = t \quad T\{\theta \circ \sigma\} = (T\{\theta\})\{\sigma\} \quad t\{\theta \circ \sigma\} = (t\{\theta\})\{\sigma\} \quad (\textit{where } \Xi \xrightarrow{\theta} \Delta \xrightarrow{\sigma} \Gamma)$$

A basic CwF includes only the bare bones of semantic models for a dependent type theory. In order to model dependent type theories of interests we need to equip them with additional structure.

We start with context extension. Let C be a CwF, and assume C_o has a terminal object $\cdot$. A **comprehension structure** $(\triangleright, \mathsf{p}, \mathsf{v})$ over C consists of, for each context $\Gamma \in C_o$ and type $T \in \mathsf{Ty}(\Gamma)$:

- A context $\Gamma \triangleright T$, the context Γ **extended by** T.
- A substitution $\mathsf{p}_T : \Gamma \triangleright T \to \Gamma$, the **weakening** of Γ by T. We say that we **weaken** a type or a term by T when we apply the corresponding substitution function for the weakening p_T.
- A term $\mathsf{v}_T \in \mathsf{Tm}_{\Gamma \triangleright T}(T)$, the **variable** we extend the context Γ with.
- Moreover, for every substitution $\theta : \Delta \to \Gamma$ and term $t \in \mathsf{Tm}_\Delta(T\{\theta\})$ there is a unique substitution $\langle \theta, t : T \rangle : \Delta \to \Gamma \triangleright T$ satisfying:

$$\mathsf{p}_T \circ \langle \theta, t \rangle = \theta \qquad \mathsf{v}_T\{\langle \theta, t \rangle\} = t$$

  We call this substitution the **extension** of the substitution θ by t. In the sequel we omit the type ascription and write $\langle \theta, t \rangle$ for $\langle \theta, t : T \rangle$ when T, as it is clear from context. Likewise, we omit the subscript on p and v when clear.

For a CwF $C = (C_o, F)$, we refer to C_o as C when doing so will cause no ambiguity.

3.2 Sections, Slices, and Dependent Types

Let C be a category and $\Gamma \in C$ an object in it. Recall the slice category C/Γ whose objects (T, d) consist of an object $T \in C$ and a morphism $d : T \to \Gamma$. Morphisms $f : (T, d) \to (S, e)$ in the slice are morphisms $f : T \to S$ that lift d through e, i.e.: $e \circ f = d$. For example, a section of Γ is a morphism out of (Γ, id) in the slice C/Γ, i.e., an object $T \in C$, and a pair of morphisms $f : \Gamma \longrightarrow T$ and $d : T \longrightarrow \Gamma$ such that $d \circ f = d$.

We can use the slices of a category C to model dependent type theory by requiring C to be locally Cartesian closed (LCCC). The intuition behind this structure is that the object (T, d) in the slice C/Γ represent types T in context Γ, and d represents the dependency of terms of this type on their context. We will not recapitulate the LCCC conditions explicitly. We will, however, spell the induced LCCC structure needed for a CwF, for two reasons. First, we indicate what we have formalized in LEAN. Second, we rely on this relationship between type families and slice objects in Section 4. It lets us use core results of sheaf theory to model dependencies in pattern matching.

Lemma 3.1. (✓ *LEAN*) *Let C be a CwF.*

- *The weakening $\mathsf{p}_T : (\Gamma \triangleright T) \longrightarrow \Gamma$ represents the type T as the slice object $((\Gamma \triangleright T), \mathsf{p}_T)$, in the sense that sections of Γ represent terms in context Γ:*

$$\mathsf{Tm}_\Gamma(T) \cong \mathsf{Hom}_{(C/\Gamma)}\Big((\Gamma, \mathrm{id}) , (\Gamma \triangleright T, \mathsf{p}_T)\Big)$$

- *The morphisms in C represent indexed types—for all $\Gamma, \Delta \in C$, $T \in \mathsf{Ty}(\Gamma)$, and $\theta : \Delta \longrightarrow \Gamma$:*

$$\mathsf{Tm}_\Delta(T\{\theta\}) \cong \mathsf{Hom}_{(C/\Gamma)}\Big((\Delta, \theta) , (\Gamma \triangleright T, \mathsf{p}_T)\Big)$$

For $t \in \mathsf{Tm}_\Gamma(T)$ we write $\bar{t} : \Gamma \longrightarrow \Gamma \triangleright T$ for $\langle id, t\rangle$, the corresponding section of p.

3.3 Semantic Type Formers and Closedness

CwFs give the core structure of type dependency, but give no indication of what types a model supports. Here we give semantic closedness conditions which postulate the existence of type and term constructors corresponding to common features of a dependent type theory.

3.3.1 Dependent Functions. We say that a CwF C supports dependent functions if it has:

- For each $S \in \mathsf{Ty}(\Gamma)$ and $T \in \mathsf{Ty}(\Gamma \triangleright S)$, a type $\Pi(S, T) \in \mathsf{Ty}(\Gamma)$;
- For each $T \in \mathsf{Ty}(\Gamma \triangleright S)$ and $t \in \mathsf{Ty}(T)$, a term $\lambda(t) \in \mathsf{Tm}(\Pi(S, T))$;
- For each $T \in \mathsf{Ty}(\Gamma \triangleright S)$, $s \in \mathsf{Tm}(S)$ and $t \in \mathsf{Tm}(\Pi(S, T))$, a term $\mathsf{App}(t, s) \in \mathsf{Tm}(T\{\langle id, s\rangle\})$;

such that the usual structural substitution rules hold, as well as the analogue of β-reduction:

$$\mathsf{App}(\lambda(t), s) = t\{\langle id, s\rangle\}$$

We could impose a version of an η rule, but this is not required for our results.

3.3.2 Equality. A CwF C supports propositional equality types [Hofmann 1997] if, for every type $T \in \mathsf{Ty}(\Gamma)$, there exists substitution-stable terms as follows:

- A type $\mathsf{Id}(T) \in \mathsf{Ty}(\Gamma \triangleright T \triangleright T\{\mathsf{p}\})$;
- A morphism $\mathsf{Refl}_T : \Gamma \triangleright T \longrightarrow \Gamma \triangleright T \triangleright T\{p\} \triangleright \mathsf{Id}(T)$, such that $\mathsf{p} \circ \mathsf{Refl} = \langle id, \mathsf{v}_T\rangle$;
- For each $S \in \mathsf{Ty}(\Gamma \triangleright T \triangleright T\{\mathsf{p}\} \triangleright \mathsf{Id}(T))$, a function (on sets) $\mathsf{J}_{T,S} : \mathsf{Tm}(T\{\mathsf{Refl}_T\}) \longrightarrow \mathsf{Tm}(S)$, such that for any $t \in \mathsf{Tm}(S\{\mathsf{Refl}_T\})$, $\mathsf{J}(t)\{\mathsf{Refl}_T\} = t$;

We have the analogues of CoverTT terms using substitution. For $T \in \mathsf{Ty}(\Gamma)$ and $s, t \in \mathsf{Tm}(T)$:

- $\mathsf{Id}(T, s, t) := \mathsf{Id}(T)\{\langle\langle id, s\rangle, t\rangle\} \in \mathsf{Tm}(\Gamma)$
- $\mathsf{Refl}_T(t) := \mathsf{v}\{\mathsf{Refl}_T \circ \bar{t}\} \in \mathsf{Tm}(\mathsf{Id}(T)\{\mathsf{p} \circ \mathsf{Refl}_T \circ \bar{t}\}) = \mathsf{Tm}(\mathsf{Id}(T, t, t))$.

We say that a CwF supports **extensional equality** when $\mathsf{Tm}(\mathsf{Id}(T, s, t)) \neq \emptyset$ iff $s = t$, such as in presheaf or set-theoretic models. Note that this condition does not require CoverTT to have an equality reflection rule: intensional equality in CoverTT can be modelled extensionally, so long as CoverTT does not include axioms like univalence that are inconsistent with equality reflection. We are mostly concerned with extensional models, but we discuss alternatives in Section 7.2.6.

3.3.3 Labelled Variants. Next we define what it means for a category to support **labelled variants**, so that we can interpret datatypes as coproducts of their constructor types. Labelled variants are simply coproducts with extra structure to mediate the type-context relationship. Assume:

- a fixed set of type constructors $\textsc{TyCon}$;
- for each $I \in \textsc{TyCon}$, a set of data constructors $\textsc{DataCon}_I$;
- for each $I \in \textsc{TyCon}$, an object $\mathsf{Params}_I \in C$ specifying types of arguments to the type constructor;
- for each $D^I \in \textsc{DataCon}_I$ a type $\mathsf{Fields}_{D^I} \in \mathsf{Ty}(\mathsf{Params}_I)$ specifying the types of arguments to the data constructor.

The last condition will usually rely on having some sort of dependent pair to encode multiple fields. Then a CwF supports the **labelled variants** for I if there exists some $\mathsf{TyCon}_I \in \mathsf{Ty}(\mathsf{Params}_I)$, such that for each $\theta : \Gamma \longrightarrow \mathsf{Params}_I$, there exists an isomorphism

$$\iota : \Gamma \triangleright \mathsf{TyCon}_I\{\theta\} \cong \coprod_i (\Gamma \triangleright \mathsf{Fields}_{D^I_i}\{\theta\}) \qquad \text{such that } \forall i, \mathsf{p}_{\mathsf{Fields}_{D^I_i}\{\theta\}} = \mathsf{p}_{\mathsf{TyCon}_I\{\theta\}} \circ \iota^{-1} \circ \mathsf{inj}_i$$

That is, projecting out the context of from the fields' type is the same as converting into TyCon_I with ι and then projecting the context. Then, for each $t \in \mathsf{Tm}(\mathsf{Fields}_{D^I_i}\{\theta\})$, the arrow $\iota^{-1} \circ \mathsf{inj}_i \circ \bar{t}$ is a section of $\mathsf{p}_{\mathsf{TyCon}_I\{\theta\}}$, and hence denotes a term in $\mathsf{Tm}(\mathsf{TyCon}_I\{\theta\})$. We call this term $\mathsf{DataCon}_{D^I_i}(t)$, since it denotes the ith data constructor applied to field values t.

3.4 Pattern Matching

Here we give a semantic presentation of $\textsc{Cover}$TT-style pattern matching. In the CwF framework, we can follow the definition of $\textsc{Cover}$TT. This serves as a statement of what we need to define pattern matching semantically. We show how to fulfill those requirements in Section 4.

Consider a semantic coverage relation $\triangleright$ whose elements are sets containing arrows into Δ. We say C supports matching over the semantic coverage $\triangleright$ if, for every, $\Delta : C$, $T \in \mathsf{Ty}(\Delta)$, index set $\mathcal{I}$, covering $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i \triangleright \Delta$ for $i \in \mathcal{I}$, branch results $t_i \in \mathsf{Tm}(T\{\theta_i\})$ for $i \in \mathcal{I}$, and scrutinee $\theta : \Gamma \longrightarrow \Delta$, there exists a term $\mathsf{match}_{\overrightarrow{(\Delta_i,\theta_i,t_i)}^i}(\theta) \in \mathsf{Tm}(T\{\theta\})$ such that:

- for any $\sigma_i : \Gamma \longrightarrow \Delta_i$, if $\theta = \theta_i \circ \sigma_i$, then $\mathsf{match}_{\overrightarrow{(\Delta_i,\theta_i,t_i)}^i}(\theta) = t_i\{\sigma_i\}$;
- The above choice commutes with substitution, i.e., for $\theta_1 : \Gamma_1 \longrightarrow \Gamma_2$ and $\theta_2 : \Gamma_2 \longrightarrow \Delta$, we have $\mathsf{match}_{\overrightarrow{(\Delta_i,\theta_i,t_i)}^i}(\theta_2 \circ \theta_1) = (\mathsf{match}_{\overrightarrow{(\Delta_i,\theta_i,t_i)}^i}(\theta_2))\{\theta_1\}$;

To see this concretely, consider the head function from Section 2.4 in the CwF structure of $\textsc{Cover}$TT. The scrutinee type Δ is $(A : \mathcal{U})(n : \mathbb{N})(x : \mathsf{Vec}\ A\ n)$. The cover is the singleton $\{\cdot \fatsemi (m+1) \fatsemi \mathsf{Cons}\ m\ h\ t\ \mathtt{refl}_{m+1}\}$, where syntactic extension $\fatsemi$ corresponds to $\langle \_, \_ \rangle$. This pattern corresponds to θ_1. The pattern context $\Delta_1 := \cdot, (m : \mathbb{N})(h : A), (t : \mathsf{Vec}\ A\ m)$, so the pattern in the cover corresponds to an arrow $\cdot, (m : \mathbb{N}), (h : A), (t : \mathsf{Vec}\ A\ m) \longrightarrow \cdot, (A : \mathcal{U}), (n : \mathbb{N}), (x : \mathsf{Vec}\ A\ n)$. There is one branch, whose result t_1 is $h : A$ (which matches the overall result because the result type is not dependent). Finally, the scrutinees are the variables $\cdot \fatsemi n \fatsemi x$, which have context-type Δ in the empty context.

The pattern matching condition says that, if $\triangleright$ is supported by a model of $\textsc{Cover}$TT, and $\{\cdot \fatsemi (m+1) \fatsemi \mathsf{Cons}\ m$
then there exists a term $\mathsf{match}_{\Delta_1,\theta_1,t_1}$ such that, for any m, h, t, we have $[\cdot \fatsemi (m{+}1) \fatsemi \mathsf{Cons}\ m\ h\ t\ \mathtt{refl}_{m+1}/\Delta]\mathsf{match}$
t. In the case of our term model, this is given by the pattern match:

$$\begin{array}{l}\mathtt{case}\ (x : \cdot \fatsemi n \fatsemi x\ :\ \cdot, (n' : \mathbb{N}), (x' : \mathsf{Vec}\ A\ n'))\ \mathtt{to}\ A\ \mathtt{of}\ \{ \\ \qquad \cdot, (m : \mathbb{N})(h : A), (t : \mathsf{Vec}\ A\ m). \cdot \fatsemi (m+1) \fatsemi \mathsf{Cons}\ m\ h\ t\ \mathtt{refl}_{m+1} \Rightarrow h\}\end{array}$$

$$[\![\cdot]\!] = \mathbb{1} \qquad [\![\Gamma, (x : T)]\!] = [\![\Gamma]\!] \rhd [\![T]\!]$$

$$[\![\Gamma \vdash \cdot \Rightarrow \cdot]\!] = !_{[\![\Gamma]\!]} : \Gamma \to \mathbb{1} \qquad [\![\Gamma \vdash \mathbf{s} \,\mathring{;}\, t \Rightarrow \Delta, (x : T)]\!] = \langle [\![\mathbf{s}]\!], [\![t]\!] \rangle : [\![\Gamma]\!] \to [\![\Delta]\!] \rhd [\![T]\!]$$

$$[\![\Gamma \vdash \mathsf{case}\ (\mathbf{t_{scrut}} : \Xi)\ \mathsf{to}\ T_{motive}\ \mathsf{of}\ \{\overrightarrow{\Delta_i.\,\mathbf{s}_i \Rightarrow t_i}^{i}\}]\!] := \mathsf{match}_{\overrightarrow{([\![\Delta_i]\!],[\![\mathbf{s}_i]\!],[\![t_i]\!])}^{i}}([\![\mathbf{t_{scrut}}]\!]) \in \mathsf{Tm}(T\{[\![\mathbf{t_{scrut}}]\!]\})$$

$$\qquad \textit{when } [\![t_i]\!] \text{are defined for all } i \text{ and } \{[\![\mathbf{s}_i]\!]\}_i \gtrdot [\![\Xi]\!]$$

$$[\![\Gamma \vdash \mathsf{case}\ (\mathbf{t_{scrut}} : \Xi)\ \mathsf{to}\ T_{motive}\ \mathsf{of}\ \{\overrightarrow{\Delta_i.\,\mathbf{s}_i \Rightarrow t_i}^{i}\}]\!]\ \textit{undefined otherwise}$$

Fig. 4. Model of Pattern Matching in CoverTT

However, in other models, there may not be an obvious way to form $\mathsf{match}_{\Delta_1,\theta_1,t_1}$. We provide a general way of forming such a term in Section 4.

3.5 Model compatibility and soundness

The correspondence between the type formers we have introduced and the constructs of CoverTT is fairly direct, but we must account for some technical details to make the connection formal.

Not every possible syntactic cover of CoverTT leads to a non-trivial model. For example, if $\{(x : \mathbb{0}).\, \mathsf{inl}\ x : (\mathbb{0} + \mathbb{N})\} \gtrdot (\mathbb{0} + \mathbb{N})$, then we can write

$$\mathsf{bad} : \mathbb{0} := \mathsf{case}\ (\mathsf{inr}\ 3 : (\mathbb{0} + \mathbb{N}))\ \mathsf{to}\ \mathbb{0}\ \mathsf{of}\ \{(x : \mathbb{0}).\, (\mathsf{inl}\ x) \Rightarrow x\}$$

If we have a CwF structure on a category C supporting functions, labelled variants, and pattern matching in the sense of Section 3.4 with a coverage relation $\gtrdot$, then we can soundly model CoverTT in C so long as the syntactic $\gtrdot$ is compatible with the semantic $\gtrdot$.

In Fig. 4 we define partial translations $[\![\Gamma]\!] \in C$, $[\![\Gamma \vdash T]\!] \in \mathsf{Ty}(\Gamma)$, $[\![\Gamma \vdash t : T]\!] \in \mathsf{Tm}_{\Gamma}(T)$. We omit the cases other than pattern matching, as they are standard. The translation for pattern matches checks if the patterns translate to a semantic cover, which is well founded because each pattern is syntactically smaller than the entire match.

We then have the following soundness result, which characterizes how the syntactic coverage must correspond to a semantic coverage supporting pattern matching.

Theorem 3.2 (soundness). *If, for every syntactic cover $\{\Delta_i \vdash \mathbf{s}_i \Rightarrow \Xi\}_i \gtrdot \Xi$ in CoverTT we have $[\![\Delta_i]\!]$ and $[\![\mathbf{s}_i]\!]$ defined, and $\{[\![\mathbf{s}_i]\!]\}_i \gtrdot [\![\Xi]\!]$, then $[\![\_]\!]$ is total on terms/types/environments/contexts that are well typed with respect to $\gtrdot$. Moreover, the model is sound with respect to definitional equality.*

Proof. If all covers are compatible, then straightforward induction shows that the undefined case never arises on well-typed terms. The argument follows the standard CwF model of type theory, except for pattern matching, where the equations from Section 3.4 directly satisfy Eq-Match. □

4 COVERAGES AND SHEAVES TO MODEL COVERTT

Theorem 3.2 lists conditions that ensure we can soundly interpret CoverTT. What categories and syntactic and semantic coverages can we find that fulfill our criteria?

In this section we connect some core concepts of sheaf theory to pattern matching. Specifically, we provide a sufficient condition for when a coverage on a category C admits semantic pattern matching as in Section 3.4.

4.1 Coverages and Sheaves

Alongside our contribution we provide a brief introduction to sheaves and sites for completeness. Systematic overviews of sheaves are given, for example, by Johnstone [2003, C2] or MacLane and Moerdijk [1992].

4.1.1 Coverages and Sites. A **sheaf-theoretic coverage** on a category C is, for each $\Delta \in C$, a set of subsets of $\mathsf{Hom}_C(\_, \Delta)$, called **covers**, which fulfill the following closure condition [Johnstone 2003, C2.1.1]: For each cover $\{f_i : \Gamma_i \to \Delta\}_{i\in 1\ldots n}$, and other morphism $g : \Xi \to \Delta$ there exists a Ξ-cover $\{h_j : \Xi_j \to \Xi\}_{j\in 1\ldots m}$ such that, for each j, there exists an f_i such $g \circ h_j$ lifts along f_i. That is, there exists a k_j making the diagram to the right commute.

$$\begin{tikzcd}
\Xi_j \arrow[r, "k_j"] \arrow[d, "h_j"] & \Gamma_i \arrow[d, "f_i"] \\
\Xi \arrow[r, "g"] & \Delta
\end{tikzcd}$$

This condition is weaker than requiring covers to be closed under pullback by any arrow. In particular, we do not require C to have all pullbacks. If C does have all pullbacks, one can saturate the coverage so that the property of being a cover is preserved under pullbacks.

A sheaf-theoretic coverage on C provides a coverage J for each object $\Delta \in C$. For every non-empty cover in J, all arrows share a common codomain, and it is then clear to which object a cover belongs. A **site** (C, J) is a category C equipped with a coverage J.

We refer to "sheaf theoretic coverages" specifically to distinguish them from coverages in the sense of Sections 2 and 3, i.e. the sets of patterns that we allow. Both denote the sets of morphisms/patterns that cover a given context, but we don't require pattern coverages to fulfill the sheaf-theoretic closure conditions. In Section 7.2.6 we show that a constructive syntactic model cannot fulfill them.

4.1.2 Sheaves. The next conceptual tools we need are those of a presheaf and a sheaf. A **presheaf** P is a functor $P : C^{\mathrm{op}} \to \mathbf{Set}$. Sheaf theorists think of presheaves as abstract collection of functions/terms, with $F\Delta \in \mathbf{Set}$ being the set of functions out of Δ/terms in context Δ. A **sheaf** is a collection that 'thinks' all pattern matches over every cover uniquely defines a term. Formulating this concept precisely involves multiple nested quantifiers, and we break it down in stages.

For any presheaf P and cover $\overline{\theta_i : \Delta_i \to \Delta} \in J$, a **matching family** is a collection $x_i \in P(\Delta_i)$ such that for every $\sigma : \Xi \to \Delta_i$ and $\sigma' : \Xi' \to \Delta_j$, if $\theta_i \circ \sigma = \theta_j \circ \sigma'$, then $P(\sigma)(x_i) = P(\sigma')(x_j)$. I.e., a matching family assigns a P value for all arrows in a cover, while agreeing in overlapping cases.

An **amalgamation** of a matching family $\overrightarrow{x_i}^i$ over $\overline{\theta_i : \Delta_i \to \Delta} \in J$ is an $x \in P\Delta$ such that $P(\theta_i)(x) = x_i$, i.e., it is a value in the covered object that is compatible with the matching family.

A **sheaf** on (C, J) for a cover $\overline{\theta_i : \Delta_i \to \Delta} \in J$ is a presheaf P such that every matching family has a unique amalgamation. A presheaf is a J-sheaf, or just a sheaf, when it is a sheaf for each cover in J. It is in this way that a J-sheaf is a presheaf that 'thinks' all covers admit pattern-matching.

Let $\mathbf{y} : C \to (C^{\mathrm{op}} \to \mathbf{Set})$ denote the **Yoneda embedding** that maps each $\Gamma \in C$ to the presheaf $\mathsf{Hom}_C(\_, \Gamma)$. A coverage is **subcanonical** when for every $\Delta \in C$, $\mathbf{y}\Delta$ is a sheaf. There is a largest such coverage $J_{canonical}$—the **canonical coverage**. We say a cover is **canonical** when it is in the canonical coverage. Every representable is a sheaf for a canonical cover, though in Section 7.2.6 we show that the converse is not true: there are models which support pattern matching, so every representable is a sheaf for each allowed pattern, but where the allowed patterns do not fulfill the necessary conditions to be a sheaf-theoretic coverage.

4.2 Pattern Matching via Sheaves

The similarity between amalgamation and pattern matching is apparent, and was informally established by Coquand [1992]: since morphisms in C correspond to substitutions (sequences of terms) in CoverTT, the sheaf condition gives a way to merge arrows (branches) with the same codomain (return type). However, to model dependent pattern matching, we need to handle the dependency of the branch result type on the scrutinee's value. Thankfully, slices give us the tools to model type dependency, and sheaf theory lets us convert subcanonical coverages on a category to coverages on a slice. The key properties, which we have mechanized in Lean, are as follows (see, e.g. Johnstone [2003, C2.2.17]):

Theorem 4.1. (✓ lean) *If (C, J) is a subcanonical site, then for $\Gamma \in C$, the site $(C/\Gamma, J_\Gamma)$ is subcanonical, where we define $\{f_i : (\Delta_i, \theta_i) \longrightarrow (\Xi, \sigma)\}_i \in J_\Gamma$ if and only if $\{f_i : \Delta_i \longrightarrow \Xi\}_i \in J$. Specifically, if $\{f_i : \Delta_i \longrightarrow \Gamma\}_i$ is canonical, then $\{f_i : (\Delta_i, f_i) \longrightarrow (\Gamma, id)\}_i \in J_\Gamma$ is too.*

These properties are related to the **fundamental theorem of topos theory**, which says that a slice of a sheaf category is equivalent to a category of sheaves over the slice.

We now have what we need to state and prove the main result of this section: a criterion ensuring that a coverage can model pattern matching. The following theorem has been mechanized in the Lean 4 theorem prover; work is underway to mechanize the soundness of the model and the coverage building rules of Section 5.

Theorem 4.2. (✓ lean) *Consider a CwF C and, for each $\Delta \in C$, a relation $\{\theta_i\}_i \gtrdot \Delta$ where the θ_i are disjoint monomorphisms into Δ. If all covers in $\gtrdot$ are canonical, then C supports pattern matching (in the sense of Section 3.4).*

Proof. Let $(C, J_{canonical})$ be a canonical site with a CwF structure and a relation $\gtrdot \subseteq J_{canonical}$. Consider a scrutinee type $\Delta : C$, dependent result type $T \in \mathsf{Ty}(\Delta)$, canonical cover $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i \gtrdot \Delta$ of non-overlapping monos, branch results $t_i \in \mathsf{Tm}(T\{\theta_i\})$, and scrutinee $\theta : \Gamma \longrightarrow \Delta$. To construct $\mathsf{match}_{\overrightarrow{(\Delta_i, \theta_i, t_i)}^i}(\theta) \in \mathsf{Tm}(T\{\theta\})$, we build a term $t'_{match} \in \mathsf{Tm}_\Delta(T\{id\})$ with which we compose the scrutinee θ. We will show how the sheaf condition corresponds to the pattern match.

Matches as Arrows. Recall from Lemma 3.1 that $\mathsf{Tm}_\Delta(T\{\theta\}) \cong \mathsf{Hom}_{(C/\Gamma)}((\Delta, \theta), (\Gamma \rhd T, \mathsf{p}_T))$. To find a term in $\mathsf{Tm}_\Delta(T\{id\})$, we use an arrow in $\mathsf{Hom}_{(C/\Delta)}((\Delta, id), (\Delta \rhd T, \mathsf{p})) = \mathsf{y}(\Delta \rhd T, \mathsf{p})(\Delta, id)$.

Pattern Sets as Slice Covers. The patterns $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i{}^i$ correspond to a canonical C-cover by our premise, so by Theorem 4.1 the cover $\{\theta'_i : (\Delta_i, \theta_i) \longrightarrow (\Delta, id)\}_i$ is canonical in C/Δ.

Branches as Matching Families. The branch results of the pattern match form a matching family for $\mathsf{y}((\Gamma \rhd T, \mathsf{p}))$. Our branches are $\overrightarrow{t_i \in \mathsf{Tm}_{\Delta_i}(T\{\theta_i\})}^i$. By Lemma 3.1, this family yields a sequence $\overrightarrow{x_i \in \mathsf{Hom}_{(C/\Delta)}((\Delta_i, \theta_i), (\Delta \rhd T, \mathsf{p}))}^i$, i.e., $\overrightarrow{x_i \in \mathsf{y}((\Delta \rhd T, \mathsf{p}))((\Delta_i, \theta_i))}^i$, which is a matching family for the presheaf $\mathsf{y}(\Delta \rhd T, \mathsf{p})$ and the cover $\{\theta_i : (\Delta_i, \theta_i) \longrightarrow (\Delta, id)\}_i$.

Amalgamating Branches. Because the cover is canonical, then $\mathsf{y}(\Delta \rhd T, \mathsf{p})$ is a sheaf for it. The sheaf condition states that the above matching family has an amalgamation $x \in \mathsf{y}(\Delta \rhd T, \mathsf{p})(\Delta, id)$, such that $\theta_i \circ x = x_i$. So Lemma 3.1 yields a term $t'_{match} \in \mathsf{Tm}(T\{id\})$ such that $t'_{match}\{\theta_i\} = t_i$.

Equations and the Scrutinee. Finally, given a scrutinee $\theta : \Gamma \longrightarrow \Delta$, we choose $t'_{match}\{\theta\}$ as $\mathsf{match}_{\overrightarrow{(\Delta_i, \theta_i, t_i)}^i}(\theta) \in \mathsf{Tm}(T\{\theta\})$. It is in $\mathsf{Tm}_\Gamma(T\{\theta\})$, so it has the correct type. It satisfies the requisite equations. Indeed, since $t'_{match}\{\theta_i\} = t_i$, whenever $\theta = \theta_i \circ \sigma_i$ for some σ_i, we have $t'_{match}\{\theta\} = t'_{match}\{\theta_i \circ \sigma_i\} = (t'_{match}\{\theta_i\})\{\sigma_i\} = t_i\{\sigma_i\}$ just as Section 3.4 requires. For substitution, given $\theta = \theta_2 \circ \theta_1$, we have $t'_{match}\{\theta_2 \circ \theta_1\} = (t'_{match}\{\theta_2\})\{\theta_1\} = \mathsf{match}_{\overrightarrow{(\Delta_i, \theta_i, t_i)}^i}(\theta_2)\{\theta_1\}$. □

The above construction lets us model pattern matching for any canonical cover, where the motive type corresponds to a representable sheaf. If J is subcanonical, then every representable is a sheaf, so we can define dependent pattern matching for any motive type. Moreover, the canonical coverage contains the covers from every subcanonical coverage, so it suffices that each allowed pattern set is a canonical cover. The disjointness and injectivity conditions ensure that the branches of a match follow the sheaf-theoretic definition of a matching family. One could instead require that branches agree on their overlap [Cockx et al. 2014b], which is suited to matching on real numbers [Sherman et al. 2018].

We conclude this section by recalling a property characterizing subcanonical covers [Johnstone 2003, C2.1.11], which we will utilize in Section 5.

Theorem 4.3. *A set of arrows $\{\theta_i : U_i \longrightarrow U\}_i$ is canonical for C if and only if, for every $\sigma : V \longrightarrow U$ and every object $T \in C$, the presheaf $\mathbf{y}(T)$ is a sheaf for the pulled back family $\{\sigma^*\theta_i : \sigma^*U_i \longrightarrow V\}_i$.*

We can use this theorem to form basic subcanonical coverages. To build new coverages from old ones, we employ **saturation conditions:** operations on coverages which do not change which presheaves are sheaves for that coverage. This is of interest to us because the canonical coverage is invariant under every saturation: it is already the largest possible subcanonical coverage, so no covers can be added without changing its notion of sheaf. The next section gives several examples of useful saturation conditions.

5 TOOLS FOR BUILDING COVERAGES

In this section, we take the abstract canonicity condition and derive concrete rules for forming canonical covers. This section justifies the idea that, in CoverTT, we can begin with a basic set of covers for each type and obtain a language of patterns by allowing nesting, variables, and multiple scrutinees. We provide some base coverages, along with composition rules that can be used to build complex coverages from simpler ones. This recreates commonly supported features of dependent pattern matching: variables, constructors for labelled variants, pruning absurd branches, and matching on `refl` with inaccessible (dot) patterns. In Section 7.2.1 we discuss potential novel coverages that can be supported using coverage semantics.

5.1 Identity and Isomorphism

The most basic canonical covers are singletons consisting of an isomorphism. If two contexts are isomorphic, then it makes sense that moving from one to the other covers all cases.

Lemma 5.1. *A presheaf is a sheaf for a singleton cover containing an isomorphism $\iota : \Gamma \cong \Delta$, and so every isomorphism is a canonical singleton cover.* [1]

The first consequence of this fact is that the canonical coverage contains all identity arrows.

Corollary 5.2. *The singleton cover $\{id : \Gamma \longrightarrow \Gamma\}$ is canonical for any category, so a pattern consisting entirely of variables $x_1, x_2, \ldots x_n$ can be safely included in any coverage for CoverTT.*

Supporting identity arrows is the bare minimum we need for pattern matching. They are the base case out of which other patterns are built, where no discrimination or computation happens at all. Likewise, a catch-all pattern, commonly written as an underscore '_', is just an unnamed variable that does not occur in the right-hand side of the branch.

More generally, the canonical coverage contains all isomorphisms. So we can devise a sound semantics for CoverTT where any isomorphism is a valid cover of a type. This allows for operations such as rearranging variables in a dependency-respecting way or re-bracketing nested sums and products.

[1] In fact, every coverage has the same sheaves as one containing all isomorphisms. For categories with all pullbacks, sheaves are usually defined in terms of Grothendieck pretopologies, whose axioms require that isomorphisms are singleton covers.

Functions written by the programmer can even be used as patterns if they are isomorphisms, opening the door for user-defined views into a type. Of course, the existence of a sound semantics does not guarantee a language we can actually implement. Checking whether a term is a definitional isomorphism is undecidable without being explicitly given its inverse. Moreover, depending on how extensional the model is, there may be terms that are isomorphisms in the model, but are not definitional isomorphisms in CoverTT.

5.2 Coproducts and Wadler Views

With variables as patterns, the next primitive patterns we need are constructors for a datatype. As we saw in Section 3.3.3, one way to model datatypes is with labelled variants. So if C has coproducts, we can model datatype constructors as injections into a coproduct context, and we can amalgamate branches that match on all the constructors of a datatype using the universal property of a coproduct.

Labelled variants are only coproducts up to isomorphism, but we have seen that isomorphisms are always singleton covers, and below in Section 5.3 we see that composition preserves canonical covers. So it suffices to consider coproducts directly. Unfortunately, in an arbitrary category, coproduct injections are not guaranteed to form a canonical cover. Theorem 4.3 requires each representable to be a sheaf for the pullback of every cover, so we need coproducts to be stable under pullback, i.e. the pullback of a coproduct is the coproduct of pullbacks. Thankfully, pullback stability of coproducts holds if C is **Set**, a presheaf category, a topos, or any other locally cartesian closed category (LCCC). We already want C to be LCCC in order to support dependent functions.

Theorem 5.3. *Suppose C has all pullbacks and that coproducts are disjoint and stable under pullback. Then $\{\mathsf{inj}_j : \Delta_i \to \coprod_{i \in I} \Delta_i\}_j$ canonically cover $\coprod_{i \in I} \Delta_i$.*

The immediate result of this theorem is that for every inductive type, the constructors are covering for that type, so long as the inductive type is modelled as the labelled variants of its constructor types. However, it is important to realize that this theorem applies for any decomposition of a type into the coproduct of other types, regardless of whether the injections correspond to constructors or not. Such a coverage introduces the possibility for views as introduced by Wadler [1987], which act as first class pattern synonyms:

Corollary 5.4. *Consider a finite $I : \mathcal{U}$, a family $S : I \to \mathcal{U}$ and an indexed function $f : (i : I) \to S\ i \to T$ in CoverTT. Suppose we have a category C with a CwF model of CoverTT supporting pattern matching as in Section 3.4 over a coverage $\_ \triangleright \_$. If $[\![T]\!] \cong [\![\Sigma(t : T)(i : I)(s : S\ i).\, t =_T f\ i\ s)]\!]$, then there is also a model of CoverTT with coverage $(\_ \triangleright \_ \cup \{f\ i\}_i)$*

That is, if T is isomorphic to the sum of some types $S\ i$ over finite i, we can safely match on a value from T, where the ith pattern is the $S\ i$ value that it corresponds to, regardless of whether T is defined as an inductive type or the $S\ i$ are its constructor types.

5.3 Nesting and Composition

With variables and constructors as primitive covers, we now need a way to combine them to build more complex covers. Adding the composition of different coverages does not change their sheaves. Every cover has the same sheaves as the sieve it generates, i.e., the closure of the cover under precomposition with any arrow in C [Johnstone 2003, C2.1.3]. So canonical covers are closed under composition:

Theorem 5.5. *For a cover J of C, If $\{f_i : \Delta_i \to \Delta\}_i \in J$, and for each i, $\{g_{ij} : \Delta_{ij} \to \Delta_i\}_{ij} \in J$, then the sheaves of (C, J) are identical to the sheaves of $(C, J \cup \{f_i \circ g_{ij} : \Delta_{ij} \to \Delta\}_i)$. So if $\{f_i : \Delta_i \to \Delta\}_i$ is canonical, and for each i, $\{g_{ij} : \Delta_{ij} \to \Delta_i\}_{ij}$ is canonical, then $\{f_i \circ g_{ij} : \Delta_{ij} \to \Delta\}_{ij}$ is canonical.*

This allows patterns to be nested: if $\{f_i\}_i$ are covering for Δ, and for each set of variables in those covering patterns, $\{g_{ij}\}_{ij}$ is covering, then we can case split each variable in the f_i into j cases corresponding to the g_{ij} patterns, and the entire resulting set can still be covering. This property gives semantic justification for the case-split operation of the Agda and Idris editor modes, where the programmer selects a variable in a pattern match, and the pattern containing the variable is replaced by the sequence of patterns that has each possible constructor application in place of the variable.

When we combine the closure of the canonical coverage under identity (isomorphisms), sum injections, and composition, we can recreate dependent pattern matching on non-indexed datatypes. However, we see in the next sections that the language of coverages also gives us the tools to handle indexing.

5.4 Absurd Contexts

If we allow ourselves some extensionality, then we can use sheaves to model absurd branches and empty cases. Suppose that C has an initial object $\mathbb{0}$, with a unique arrow $0_\Gamma : \mathbb{0} \longrightarrow \Gamma$ for every Γ. The initial context denotes an empty or absurd context, since we can derive a term of any type from it. It turns out that we do not ever need to include branches for patterns whose contexts are empty. If we can amalgamate for a cover where one pattern has an empty context, we can amalgamate for the same cover with that arrow deleted, since any matching family for the smaller cover can be turned into one for the larger cover by adding the unique arrow out of the initial context.

Theorem 5.6. *Let c be a canonical cover with $\theta : \Delta \longrightarrow \Gamma \in c$. If there exists an arrow $\iota : \Delta \longrightarrow \mathbb{0}$, then $c \setminus \{\theta\}$ is canonical.*

This property mirrors how Agda and Idris allow for the omission of empty cases. In some cases, these languages only allow branch right-hand sides to be removed after the programmer specifies an empty pattern, marking which part of the scrutinee has an impossible type. We view this empty pattern as a syntactic aid to tell the type checker when a context is isomorphic to the initial context, so we do not directly model the empty pattern.

Like our assumption about equality, the condition of having a (strong) initial object does not hold in the term model, since not all eliminations of the empty type are definitionally equal. So long as CoverTT does not contain any axioms that specifically distinguish empty eliminations, our model is still sound.

5.5 Propositional Equality

Since isomorphisms are always canonical singleton covers (Lemma 5.1), we can create a coverage for a sufficiently-extensional equality type.

Corollary 5.7. *If $\langle \mathsf{v}, \langle \mathsf{v}, \mathsf{Refl}_A \rangle\rangle : \Gamma \triangleright A \longrightarrow \Gamma \triangleright A \triangleright A\{\mathsf{p}\} \triangleright \mathsf{Id}(A\{\mathsf{p}^2\}, \mathsf{v}, \mathsf{v}\{\mathsf{p}\})$ is an isomorphism, then $\{\mathsf{Refl}_T\}$ canonically covers $\Gamma \triangleright A \triangleright A\{\mathsf{p}\} \triangleright \mathsf{Id}(A\{\mathsf{p}^2\}, \mathsf{v}, \mathsf{v}\{\mathsf{p}\})$*

In more readable, non CwF notation: if A is isomorphic to $\Sigma(x : A)(y : A).\, x =_A y$ in the model via the projections, then $\{(x, x, \mathtt{refl}_x)\}$ can be a singleton cover for $\Sigma(x : A)(y : A).\, x =_A y$. Such an isomorphism holds if C has extensional equality, since it asserts that there is a unique, internally constructible proof of equality between two equal terms.

For a dependent match with result type $(x : A), (y : A), (pf : x =_A y) \vdash P(x, y, pf) : \mathcal{U}$, the above cover only requires we provide a branch result with type $P(x, x, \mathsf{Refl}_A)$. The variable y was replaced by x in the goal type. This captures “inaccessible” or “forced” patterns [Norell 2009], known as dot-patterns in Agda and Idris. By matching on the propositional equality, we work with refined information about the context. Here, we match y against the variable x rather than a constructor.

No branching or discrimination happening, since the cover is a singleton. Rather, x is the only possible value for y given the equality proof. Agda writes $.x$ to express this pattern. Section 5.6.2 extends this to equality proof is between arbitrary terms, rather than variables.

5.6 Pullbacks and Unification

We have seen that the canonical coverage includes isomorphisms and injections and that it allows for composition. However, the definition of a sheaf-theoretic coverage enables stability under pullback: for any coverage, a sheaf for that coverage is still a sheaf if we add the pullback of any cover by any arrow. Closure under pullback is the key condition that separates a coverage from a set of arrows. The definition in Section 4.1 is presented in the style of Johnstone [2003, C2], in a general way that does not assume the existence of pullbacks. However, when each arrow in a cover has a pullback along some morphism, we get the following saturation condition:

Theorem 5.8. *For a cover J of C, if $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i \in J$, and $g : \Gamma \longrightarrow \Delta$, where the pullback of each θ_i along g exists, then $J \cup \{\{g^*\theta_i : g^*\Delta_i \longrightarrow \Gamma\}_i\}$ has the same sheaves as J. So if $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i$ is canonical, then so is $\{g^*\theta_i : g^*\Delta_i \longrightarrow \Gamma\}_i$.*

This abstract property, known as **stability under base change**, can be exploited to build interesting covers.

5.6.1 Context Extension. Base change lets us add new scrutinees to a pattern match. In any CwF, for $\theta : \Gamma \longrightarrow \Delta$ and $T \in \mathsf{Ty}(\Delta)$, pulling back by $\mathsf{p} : \Delta \triangleright T \longrightarrow \Delta$ yields $\langle \theta \circ \mathsf{p}, \mathsf{v} \rangle$ [Hofmann 1997]. Combining this with Theorem 5.8 gives:

Theorem 5.9. *For canonical $\{\theta_i : \Delta_i \longrightarrow \Delta\}_i$ and a type $T \in \mathsf{Ty}(\Delta)$, there is also a canonical cover $\{\langle \theta_i \circ \mathsf{p}, \mathsf{v} \rangle : \Delta_i \triangleright T\{\theta_i\} \longrightarrow \Delta \triangleright T\}_i$.*

So we can build a covering pattern for a context Δ and immediately obtain a covering context on $\Delta \triangleright T$ by appending a new variable $\mathsf{v} \in \mathsf{Tm}(T\{\theta_i\})$ to each pattern in the cover. In a dependent match the new variable might have a different type in each branch: T may be indexed by variables in Δ, but each θ_i is a value for Δ with variables from Δ_i. Further case-splitting on the newly introduced variable can be achieved using composition à la Section 5.3.

5.6.2 Matching on Equality. Suppose that for some $\Gamma \in C$ and $T \in \mathsf{Ty}(\Gamma)$, and that: $\{\mathsf{Refl}_T : \Gamma \triangleright T \longrightarrow \Gamma \triangleright T \triangleright T\{$
is canonical. As in Section 5.5, such a property holds for an extensional model. With such a cover on equality, we can apply the base change theorem, the CwF laws, and the properties of equality (Section 3.3.2) to obtain a cover on contexts containing equalities by matching:

Theorem 5.10. *Suppose C has all pullbacks. Consider $\Gamma \in C$ with $T \in \mathsf{Ty}(\Gamma)$, $t_1, t_2 \in \mathsf{Tm}(T)$. Then pulling back Refl_T by $\langle \langle \overline{t_1\{\mathsf{p}\}, t_2\{\mathsf{p}\}} \rangle, \mathsf{v} \rangle$ yield a context Δ and an arrow $\langle \theta, \mathsf{Refl}_T(t_{12}) \rangle$, where $\theta : \Delta \longrightarrow \Gamma$, $t_{12} \in \mathsf{Tm}(T\{\theta\})$, and $t_1\{\theta\} = t_2\{\theta\} = t_{12}$.*

Moreover, if $\{\mathsf{Refl}_T\}$ is canonical, then $\{\langle \theta, \mathsf{Refl}_T(t_{12}) \rangle : \Delta \longrightarrow \Gamma \triangleright \mathsf{Id}(T, t_1, t_2)\}$ is canonical.

For the intuition behind this, consider the pullback square to the right, translated to CoverTT-style notation for clarity. First, because the square commutes, we know that θ is a substitution that equates (t_1, t_2, pf) and $(x, x, \texttt{refl}_x)$, i.e., it is a **unifier** of t_1 and t_2. Since a pullback is a limit, it is universal, so any other unifiers for t_1 and t_2 necessarily factor through θ. Thus, it is the **most general unifier** for t_1 and t_2. This is precisely what the usual

$$\begin{tikzcd}
\Delta \arrow[r, "\theta \fatsemi t_{12}"] \arrow[d, "\theta \fatsemi \texttt{refl}_{t_{12}}"] & \Gamma, (x : T) \arrow[d, "(x \fatsemi x \fatsemi \texttt{refl}_x)"] \\
\Gamma, (pf : x =_T y) \arrow[r, "(t_1 \fatsemi t_2 \fatsemi s)"] & \Gamma, (x : T), (y : T), (x =_T y)
\end{tikzcd}$$

rule for pattern matching on equality uses: it unifies the two sides of the equality, treating syntactic variables as unification variables, and generates a substitution that is then applied to the goal. The context Δ consists of the variables that were in common between t_1 and t_2 which remain free in the unification t_{12}. In the case that t_1 and t_2 do not unify, then the pullback is an arrow out of an initial context, and the branch can be omitted completely (because absurd covers can be omitted, as in Section 5.4).

6 EXAMPLE: FOLDING WITHOUT A STARTING VALUE

We now have specified everything we need (sans recursion) for feature-parity with the original presentation of dependent pattern matching by Coquand [1992]. Our sheaf-centric view generalizes the elaboration process of Goguen et al. [2006], but directly within the model instead of as a syntactic elaboration. Constructors for a coproduct form a cover and variables form a singleton cover, acting as the basis from which other covers are generated. Covers can be composed, extended, refined by matching on an equality, or pared down by pruning absurd branches. Indexed data types can be handled using fording and matching on equality.

To see a non-trivial example of how to build a cover in the canonical coverage, consider the foldr_1 function found in the Agda standard library [Documentation for Agda 2024].

$$\mathsf{foldr}_1 : (A \to A \to A) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A$$

$$\mathsf{foldr}_1\ f\ (\mathsf{cons}\ x\ \mathsf{nil})\ = x \qquad | \qquad \mathsf{foldr}_1\ f\ (\mathsf{cons}\ x\ (\mathsf{cons}\ y\ ys)\ = f\ x\ (\mathsf{foldr}_1\ f\ (\mathsf{cons}\ y\ ys))$$

Because the argument vector has length at least one, the case for nil can be omitted. The base case is then a vector of length one, and the inductive case is a vector of length two or more.

Assume an extensional CwF model of CoverTT in a LCCC C where inductive types are labelled variants. We show how the patterns for foldr_1 are in the canonical coverage for C, and hence foldr_1 can be modelled. Note that because labelled variants are defined in terms of isomorphism, we do not preclude initial algebra semantics for modelling the self-reference part of inductive types.

6.1 Translating to CoverTT

First, we translate the function to CoverTT-style by making the length argument explicit and replacing the indexed constructors with ones taking explicit equality proofs. The datatype becomes:

$$\begin{aligned}
&\texttt{data}\ \mathsf{Vec}\ (A : \mathcal{U}) : (m : \mathbb{N}) \to \mathcal{U}\ \texttt{where} \\
&\quad \mathsf{nil} : (n = 0) \to \mathsf{Vec}\ A\ n \\
&\quad \mathsf{cons} : (m : \mathbb{N}) \to A \to \mathsf{Vec}\ A\ m \to n = m + 1 \to \mathsf{Vec}\ A\ n.
\end{aligned}$$

We also abstract out the recursive calls, since we have not included them in CoverTT and have not required that our model category C support them. Despite using recursion, foldr_1 is an ideal example because it is not contrived, and uses all the main saturation conditions we developed in Section 5. Since foldr_1 is decreasing in the length of the lists, techniques for modelling the recursion are well known and orthogonal to our contribution.

$$\begin{aligned}
&\mathsf{foldr}_1 : (n : \mathbb{N}) \to (A \to A \to A) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \\
&\qquad\qquad \to (\mathsf{self} : (A \to A \to A) \to \mathsf{Vec}\ A\ (\mathsf{suc}\ n) \to A) \to A \\
&\mathsf{foldr}_1\ 0\ f\ (\mathsf{cons}\ 0\ x\ (\mathsf{nil}\ \mathsf{Refl})\ \mathsf{Refl})\ \mathsf{self} = x \\
&\mathsf{foldr}_1\ (m+1)\ f\ (\mathsf{cons}\ (m+1)\ x\ (\mathsf{cons}\ m\ y\ ys\ \mathsf{Refl})\ \mathsf{self} = f\ x\ (\mathsf{self}\ f\ (\mathsf{cons}\ y\ ys\ \mathsf{Refl})\ \mathsf{Refl})
\end{aligned}$$

6.2 Building the Coverage

Using CoverTT notation rather than CwF notation for clarity and space reasons, we now show how the rules of Section 5 can be used to build a cover:

$$\begin{aligned}&\{((m+1)\ f\ (\mathsf{cons}\ (m+1)\ x\ (\mathsf{nil}\ \mathsf{Refl})\ \mathsf{Refl})\ \mathsf{self}),\\ &\ ((m+1)\ f\ (\mathsf{cons}\ (m+1)\ x\ (\mathsf{cons}\ m\ y\ ys\ \mathsf{Refl})\ \mathsf{Refl})\ \mathsf{self})\}\\ &\qquad\qquad \gtrdot (n:\mathbb{N}), (f : A \to A \to A), (v : \mathsf{Vec}\ A\ (\mathsf{suc}\ n)), (\mathsf{self} : \ldots)\end{aligned}$$

- Identity (Corollary 5.2) gives that the variables $(n)(f)(v)(\mathsf{self})$ cover the scrutinee type;
- Coproduct (Corollary 5.4) has $\{(\mathsf{nil}\ eq_{nil}), (\mathsf{cons}\ m'\ x\ xs\ eq_{cons})\}$ covering Vec A q for all q;
- Composition (Theorem 5.5) gives a cover $\{(n\ f\ (\mathsf{nil}\ eq_{nil})\ \mathsf{self}), (n\ f\ (\mathsf{cons}\ m'\ x\ xs\ eq_{cons})\ \mathsf{self})\}$;
- We have $eq_{nil} : n+1 = 0$, but this type is empty, so applying the absurd rule (Theorem 5.6) gives a singleton cover $\{(n\ f\ (\mathsf{cons}\ m'\ x\ xs\ eq_{cons})\ \mathsf{self})\}$;
- $eq_{cons} : n+1 = m'+1$, so we get a pullback substitution mapping m' to n and all other variables to themselves. Applying the Refl rule (Corollary 5.7), $n\ n$ Refl is a cover of $(m' : \mathbb{N}) \triangleright (n : \mathbb{N}) \triangleright (n+1 = m'+1)$. By composition, our scrutinee context has a singleton cover $\{(n\ f\ (\mathsf{cons}\ n\ x\ xs\ \mathsf{Refl})\ \mathsf{self})\}$ in the canonical coverage;
- Again using the coproduct property for Vec with composition, we have a cover $\{(n\ f\ (\mathsf{cons}\ n\ x\ (\mathsf{nil}\ eq_{nil})\ \mathsf{Refl})\ \mathsf{self}), (n\ f\ (\mathsf{cons}\ n\ x\ (\mathsf{cons}\ m\ y\ ys\ eq_{cons})\ \mathsf{Refl})\ \mathsf{self})\}$;
- Finally, since $eq_{nil} : n = 0$ and $eq_{cons} : n = m+1$, we can apply the Refl rule for each of these proofs, along with composition, to obtain the desired cover above.

Then, we can use the sheaf condition model how $f\ x\ (\mathsf{self}\ f\ (cons\ y\ ys\ \mathsf{Refl})\ \mathsf{Refl})$ and x are amalgamated into a denotation for the entire function.

7 DISCUSSION

7.1 Related Work

Dependent pattern matching was first proposed by Coquand [1992]. While this work contains no explicit mentions of sheaf theory, it originated the idea that patterns could be thought of in terms of coverings and partitions of a space, which greatly inspired our work. The theory and practice of both pattern matching and eliminators foundational developments in proof assistants: McBride [2002] developed elimination for Lego, and later Epigram [McBride 2005]. This was extended to views and with-clauses by Mcbride and Mckinna [2004].

Goguen et al. [2006] show how pattern matching can be elaborated to primitive eliminators, and hence given semantics in any model that had semantics for eliminators. These are in turn given semantics using initial algebras [Abbott et al. 2005; Altenkirch et al. 2015]. Cockx et al. [2014a] extend this to work with univalent theories. Elaboration is similar to amalgamation using the sheaf condition, but amalgamation occurs strictly in the model. Elaborating to eliminators also handles recursion, which is not yet explicitly included in our sheaf semantics.

To our knowledge, the first explicit connection between the sheaves and pattern matching was by Sherman et al. [2018], which gave a framework for pattern-matching on real numbers using topological spaces. The thesis version of this work [Sherman 2017] generalizes the approach from topological spaces to Grothendieck topologies. Cockx et al. [2014b] give similar semantics to overlapping patterns by treating them as definitional equalities, using confluence rather than sheaves.

Our approach of using pullbacks to represent unification is related to the approach of Cockx and Devriese [2018], where unification of indices in pattern matches is expressed using explicit proof terms in the language itself. We show how one can build a match given the corresponding equalities, whereas their approach shows how to actually deduce such equalities.

7.2 Future Work

7.2.1 First-Class Pattern Synonyms. An immediate application of this work would be to implement an enhanced version of pattern synonyms in a language like Agda. Currently, Agda lets the programmer declare pattern synonyms, but each name must map to a syntactic pattern i.e., a set of nested constructor applications. Agda checks if a definition is covering by elaborating to these patterns. Our framework could be used to build direct coverage checking for pattern synonyms, so the programmer could build their own alternate, extensible covers of a type, using sheaf theory to justify their coverage. This would provide direct semantics for the user-defined views of Wadler [1987] and Mcbride and Mckinna [2004]. Further research is needed to extract a constructive procedure for amalgamating branches that can be implemented in practice.

7.2.2 Overlapping Pattern Matches. Our semantics require non-overlapping, injective patterns, but our framework suggests a way to lift this restriction. Recall that the sheaf condition only requires that a matching family agrees on the overlap between covering patterns. This suggests two ways to give semantics to overlapping patterns: by ensuring that the right-hand sides of each pattern match agree on the overlap of their left-hand sides, or by adding information to each pattern to ensure they are actually non-overlapping.

The latter approach matches current implementations: catch-all patterns are elaborated into multiple branches whose left-hand patterns are the constructors that have not yet been used. Unfortunately, to prove anything about a function defined this way, the programmer needs a proof case for each branch in the elaboration, even if they correspond to a single branch in the function as written. Our framework may support canonical covers which contain extra information preventing overlap, such as proofs that previous branches had not matched. These could be used to develop covers for matches with overlapping patterns that do not require creating additional cases during elaboration, enabling more succinct proofs about overlapping cases.

7.2.3 Inductive Datatypes and Termination Checking. When patterns are not restricted to constructors, it is not immediately apparent which recursive pattern matching functions can be soundly modelled, since pattern variables may not be structurally smaller than the patterns in which they occur. Further study is needed to devise criteria for which recursive definitions are well founded with non-constructor patterns.

7.2.4 Beyond Top-Level Matches. Our semantics only support top level pattern matches. Many of the results we used, such as the fundamental theorem of topos theory, are well suited to top-level matches but do not directly translate to terms in an arbitrary context. Additionally, if the scrutinee type of a pattern match is in a non-empty context, then matching affects not only the motive, but may refine the values or types of variables in the context on the left. These technical issues suggest a semantic theory of *telescopes*, which are objects representing extensions to a given context by some number of types, and *environments*, which extend substitutions by some number of terms.

7.2.5 With Clauses. Our approach to matching on equality proofs gives an intuition for modelling Agda-style with-clauses and views [Mcbride and Mckinna 2004], though a full account is beyond the scope of this paper. Suppose we are defining a pattern match with scrutinees of type Δ and result type $\Delta \vdash T : \mathcal{U}$, and we want to match on some intermediate expression $\Delta \vdash s : S$. There is an isomorphism $\iota : \Delta \cong \Delta, (x : S), (pf : x =_S s)$. So if we have a cover of $\{s_i : \Delta_i \to \Delta\}_i$ to match s against, we can use composition and extension to obtain a cover $\{s_i : \Delta, (pf : s_i = s) \to \Delta\}_i$. In the case that s_i and s unify, the Refl rule from above can be used to match on the equality, and in an extensional model, the goal type can be safely rewritten due to the existence of the equality proof.

7.2.6 Intensional Models. Our current approach relies on extensional models, where equality proofs correspond with equality in the model. We can define models of CoverTT in terms of canonical coverages and non-syntactic equality, and we can give a CwF term model for CoverTT because syntactic pattern matching fulfills the criteria of Section 3.4, but the term model for CoverTT cannot be described in terms of sheaf-theoretic coverages.

To show the issue, we show that $\{\mathsf{true}, \mathsf{false}\}$ is not a canonical cover of Bool for the CwF given by well-typed CoverTT terms quotiented by definitional equality. Consider a function $\mathsf{haltsInN} : \mathsf{SyntaxTree} \to \mathbb{N} \to \mathsf{Bool}$, that looks at a syntax tree of an untyped lambda calculus term and checks whether it halts in n or fewer steps. Consider also $\Omega : \mathsf{SyntaxTree}$, a representation of $(\lambda x.\, x\ x)(\lambda x.\, x\ x)$. If $\{\mathsf{true}, \mathsf{false}\}$ is a canonical cover, it is also a sieve in the canonical topology [Johnstone 2003, C2.1.8], so pulling the sieve back by $\mathsf{haltsInN}\ \Omega$ produces the set of arrows $\{h_j \circ !_{V_j} : V_j \to \mathbb{N} \mid h_j : \mathbb{1} \to \mathbb{N}\}$. This contains each arrow in $\mathbb{1} \to \mathbb{N}$, i.e., each natural number. For the cover to be canonical, for any type T there must be a way to amalgamate $\{t_j : T \mid h_j : \mathbb{1} \to \mathbb{N}\}$ into $\mathbb{N} \to T$. Then all set-theoretic infinite sequences of natural numbers could be amalgamated into type-theoretic functions $\mathbb{N} \to \mathbb{N}$, which is impossible.

The above example relies on the existence of an infinite cover. While infinite covers are allowed in sheaf theory, they do not correspond directly to pattern matches that a programmer can write down. So further exploration of finite and infinite covers may resolve the issue.

Another issue is that extensional models typically imply that all equality proofs of a given type are equal. As such our approach is incompatible with univalent theories like Homotopy or Cubical Type Theory [Cohen et al. 2018; Univalent Foundations Program 2013]. Both of these issues might be addressed by replacing sheaves with stacks or, even more generally, ∞-stacks [Lurie 2009]. These replace the strict equality of the sheaf condition with higher structure. However, the technical and theoretical overhead of switching to stacks is considerable, and utilizing them for pattern matching will be a significant undertaking.

7.2.7 Toposes and Quasi-toposes. Apart from pattern matching, the theory of sheaves plays a central role in categorical logic, since categories of sheaves over a coverage form *Grothendieck toposes*, which serve as models of constructive logic. Quasi-toposes relax the sheaf conditions to only require uniqueness of amalgamations. Future work should search for deeper connections to toposes or quasi-toposes. Two-level type theories [Annenkov et al. 2023] may yield some answers, since they describe the interactions between a model of a type theory and the category of presheaves over that model.

7.3 Conclusion

This work formalizes the connection between dependent pattern matching and the notion of sheaves over a site. We have provided a framework which is expressive enough to capture the semantics of current pattern matching implementations, while laying the groundwork for future enhancements. Our work demonstrates that elaboration to eliminators is not the only feasible semantics for dependent pattern matching, and that there is perspective to be gained from treating pattern matching as a core feature and using the lens of sheaf theory.

REFERENCES

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing Strictly Positive Types. Theor. Comput. Sci. 342, 1 (2005), 3–27. https://doi.org/10.1016/j.tcs.2005.06.002

Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor Mcbride, and Peter Morris. 2015. Indexed Containers. J. Funct. Program. 25 (Jan. 2015), e5. https://doi.org/10.1017/S095679681500009X

Danil Annenkov, Paolo Capriotti, Nicolai Kraus, and Christian Sattler. 2023. Two-Level Type Theory and Applications. Math. Struct. Comput. Sci. 33, 8 (Sept. 2023), 688–743. https://doi.org/10.1017/S0960129523000130

Yves Bertot and Pierre Castéran. 2004. Interactive Theorem Proving and Program Development. Springer-Verlag.

Edwin Brady. 2021. Idris 2: Quantitative Type Theory in Practice. In 35th Eur. Conf. Object-Oriented Program. ECOOP 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2021.9

Jesper Cockx and Dominique Devriese. 2018. Proof-Relevant Unification: Dependent Pattern Matching with Only the Axioms of Your Type Theory. J. Funct. Program. 28 (2018), e12. https://doi.org/10.1017/S095679681800014X

Jesper Cockx, Dominique Devriese, and Frank Piessens. 2014a. Pattern Matching without K. In Proc. 19th ACM SIGPLAN Int. Conf. Funct. Program. (ICFP '14). ACM, New York, NY, USA, 257–268. https://doi.org/10.1145/2628136.2628139

Jesper Cockx, Frank Piessens, and Dominique Devriese. 2014b. Overlapping and Order-Independent Patterns. In Program. Lang. Syst., Zhong Shao (Ed.). Springer, Berlin, Heidelberg, 87–106. https://doi.org/10.1007/978-3-642-54833-8_6

Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In 21st Int. Conf. Types Proofs Programs TYPES 2015 (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 69), Tarmo Uustalu (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 5:1–5:34. https://doi.org/10.4230/LIPIcs.TYPES.2015.5

Thierry Coquand. 1992. Pattern Matching with Dependent Types. In Informal Proc. Log. Framew., Vol. 92. 66–79.

Documentation for Agda. 2024. Data.Vec.Base. https://github.com/agda/agda-stdlib/blob/196766082e913de0d7cd98e3b672935a3b4528b8/src/Data/Vec/Base.agda.

Healfdene Goguen, Conor McBride, and James McKinna. 2006. Eliminating Dependent Pattern Matching. In Algebra, Meaning, and Computation: Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday, Kokichi Futatsugi, Jean-Pierre Jouannaud, and José Meseguer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 521–540. https://doi.org/10.1007/11780274_27

Martin Hofmann. 1997. Syntax and Semantics of Dependent Types. In Semantics and Logics of Computation, Andrew M. Pitts and P. Dybjer (Eds.). Cambridge University Press, Cambridge, 79–130. https://doi.org/10.1017/CBO9780511526619.004

Peter T. Johnstone. 2003. Sketches of an Elephant: A Topos Theory Compendium. Oxford University Press, Oxford, New York.

Jacob Lurie. 2009. Higher Topos Theory. Number no. 170 in Annals of Mathematics Studies. Princeton University Press, Princeton, N.J.

Saunders MacLane and Ieke Moerdijk. 1992. Sheaves in Geometry and Logic: A First Introduction to Topos Theory. Springer, New York, NY, UNITED STATES.

Conor McBride. 2000. Dependently Typed Functional Programs and Their Proofs. Ph. D. Dissertation. University of Edinburgh, UK.

Conor McBride. 2002. Elimination with a Motive. In Types Proofs Programs (Lecture Notes in Computer Science), Paul Callaghan, Zhaohui Luo, James McKinna, Robert Pollack, and Robert Pollack (Eds.). Springer, Berlin, Heidelberg, 197–216. https://doi.org/10.1007/3-540-45842-5_13

Conor McBride. 2005. Epigram: Practical Programming with Dependent Types. In Adv. Funct. Program., Varmo Vene and Tarmo Uustalu (Eds.). Springer, Berlin, Heidelberg, 130–170. https://doi.org/10.1007/11546382_3

Conor Mcbride and James Mckinna. 2004. The View from the Left. J. Funct. Prog. 14, 1 (Jan. 2004), 69–111. https://doi.org/10.1017/S0956796803004829

Ulf Norell. 2009. Dependently Typed Programming in Agda. In Proc. 4th Int. Workshop Types Lang. Des. Implement. (TLDI '09). ACM, New York, NY, USA, 1–2. https://doi.org/10.1145/1481861.1481862

Benjamin Sherman. 2017. Making Discrete Decisions Based on Continuous Values. Master of Science. Massachusetts Institute of Technology.

Benjamin Sherman, Luke Sciarappa, Adam Chlipala, and Michael Carbin. 2018. Computable Decision Making on the Reals and Other Spaces: Via Partiality and Nondeterminism. In Proc. 33rd Annu. ACMIEEE Symp. Log. Comput. Sci. ACM, Oxford United Kingdom, 859–868. https://doi.org/10.1145/3209108.3209193

The Univalent Foundations Program. 2013. Homotopy Type Theory: Univalent Foundations of Mathematics. https://homotopytypetheory.org/book, Institute for Advanced Study.

P. Wadler. 1987. Views: A Way for Pattern Matching to Cohabit with Data Abstraction. In Proc. 14th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang. (POPL '87). Association for Computing Machinery, New York, NY, USA, 307–313. https://doi.org/10.1145/41625.41653