Focusing on Pattern Matching

Neelakantan R. Krishnaswami
Carnegie Mellon University
neelk@cs.cmu.edu

Abstract

In this paper, we show how pattern matching can be seen to arise from a proof term assignment for the focused sequent calculus. This use of the Curry-Howard correspondence allows us to give a novel coverage checking algorithm, and makes it possible to give a rigorous correctness proof for the classical pattern compilation strategy of building decision trees via matrices of patterns.

Categories and Subject Descriptors F.4.1 [*Mathematical Logic*]: Lambda Calculus and Related Systems

Keywords Focusing, Pattern Matching, Type Theory, Curry-Howard

1. Introduction

From the point of view of the semanticist, one of the chief attractions of functional programming is the close connection of the typed lambda calculus to proof theory and logic via the Curry-Howard correspondence. The point of view of the workaday programmer seems, at first glance, less exalted — one of the most compelling features in actual programming languages like ML and Haskell is the ability to analyze structured data with pattern matching. But pattern matching, though enormously useful, has historically lacked the close tie to logic that the other core features of functional languages possess. The Definition of Standard ML (Milner et al. 1997), for example, contains a suggestion in English that it would be desirable to check coverage, with no clear account of what this means or how to accomplish it.

Our goal is to rectify this discrepancy, and show that the semanticist ought to be just as interested in pattern matching as the programmer. We show that pattern matching has just as strong a logical interpretation as everything else in functional programming, and that filling in this part of the Curry-Howard correspondence enables simple correctness proofs of parts of the compiler (such as the coverage checker and the pattern compiler) that required considerable insight and creativity before.

Specifically, our contributions are:

- First, we give a proof term assignment for a focused sequent calculus, which naturally gives rise to pattern matching. Then, we show how to extend this calculus to properly model features of ML-like pattern languages such as as-patterns, or-patterns, incompleteness, and the left-to-right priority ordering of ML-

POPL '09 21-23 January 2009, Savannah, Georgia, USA.
Copyright © 2009 ACM [to be supplied]…$5.00

style pattern matching. This calculus also extends easily to encompass features like recursive and existential types.

- Second, we give a simple inductive characterization of when a pattern is exhaustive and non-redundant, and prove that this algorithm is sound. This is of interest since it is a coverage test that does not involve examining the output of a pattern compilation algorithm.
- Third, we reconstruct the classical matrix-based method of compiling patterns into decision trees in terms of our formalism, and prove its correctness. This correctness result is stronger than prior results, since it is based directly upon the language's semantics.

2. An Introduction to Focusing

2.1 The Focused Sequent Calculus

A rule of a logical system is called *invertible* when the conclusion of the rule is strong enough to imply the premises; that is, when the rule can be read bidirectionally. For example, the right rule for implication in the sequent calculus is invertible:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}$$

We can give a proof that this rule is invertible by showing that the conclusion lets us deduce the premise:

$$\dfrac{\dfrac{\mathbf{\Gamma \vdash A \to B}}{\Gamma, A \vdash A \to B}\,\text{Weak} \qquad \dfrac{\dfrac{}{\Gamma, A \vdash A}\,\text{Hyp} \qquad \dfrac{}{\Gamma, A, B \vdash B}\,\text{Hyp}}{\Gamma, A, A \to B \vdash B}\,{\to}\text{L}}{\Gamma, A \vdash B}\,\text{Cut}$$

Since the conclusion can be deduced from the premises, and the premises follow from the conclusion, this means that applying an invertible rule cannot change the provability of a sequent — the same information is available. (In contrast, a rule like sum-introduction is *not* invertible, since the knowledge that $A+B$ holds is less informative than knowing that, say, A holds.)

This is problematic for applications involving proof search, such as theorem proving. The fact that we can apply invertible rules at any time means that there are many equivalent proofs that differ only in the order that inversion steps are made in, which increases the size of the theorem prover's search space with no benefit. Andreoli introduced the concept of focusing (Andreoli 1992) as a technique to reduce the nondeterminism in the sequent calculus. First, he observed (as had others (Miller et al. 1991)) that applications of invertible rules could be done eagerly, since inversion does not change provability. Then, he observed that each connective in linear logic was either invertible on the left and non-invertible on the right (the *positive* types), or invertible on the right and non-invertible on the left (the *negative* types). Finally, he made the remarkable observation that in a fully inverted sequent (i.e., one in which no further inversions are possible), it is possible to select a single hypothesis (on either the left or the right) and then eagerly try to prove it, *without losing completeness*. This fact explains the name focusing; one can "focus on" a hypothesis without losing completeness. We give a version of the focused sequent calculus for intuitionistic logic below. First, we group the connectives.

$$\begin{array}{llll}
\text{Types} & A & ::= & X \mid 1 \mid A \times B \mid A \to B \mid 0 \mid A + B \\
\text{Positives} & P & ::= & X \mid 1 \mid A \times B \mid 0 \mid A + B \\
\text{Negatives} & N & ::= & X \mid A \to B
\end{array}$$

The types are unit 1, products $A \times B$, void 0, sums $A + B$, function spaces $A \to B$, and atomic types X. Sums and products[1] are positive, and function space is negative. Atoms are treated as either polarity, as convenient. This system has four judgements. First, we have a right focus phase $\Gamma \vdash A$, in which we try to prove a positive formula on the right. As usual, Γ is taken to be a list of hypotheses, with the rules presented in such a way as to make the usual structural properties (exchange, weakening and contraction) admissible.

$$\boxed{\Gamma \vdash A}$$

$$\frac{}{\Gamma \vdash 1}\ 1\mathrm{R} \qquad \frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \times B}\ \times\mathrm{R} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A + B}\ +\mathrm{R1}$$

$$\frac{\Gamma \vdash B}{\Gamma \vdash A + B}\ +\mathrm{R2} \qquad \frac{\Gamma; \cdot \vdash N}{\Gamma \vdash N}\ \textsc{BlurR}$$

Since the leaves of a positive proposition can contain negative formulas, the BLURR rule lets us transition to the two-context right inversion judgement $\Gamma; \Delta \vdash A$, in which a negative proposition on the right is inverted. Since the only negative connective is implication, this means moving the left-hand-sides of arrows $A \to B$ into the context Δ, which can contain arbitrary positive or negative formulas. The Δ context is *ordered* – the structural rules of exchange, contraction and weakening are *not* permitted in this context.

$$\boxed{\Gamma; \Delta \vdash A}$$

$$\frac{\Gamma; \Delta, A \vdash B}{\Gamma; \Delta \vdash A \to B}\ {\to}\mathrm{R} \qquad\qquad \frac{\Gamma; \Delta \rhd P}{\Gamma; \Delta \vdash P}\ \textsc{BlurL}$$

Once the right-hand-side is positive once again, we have to continue with inversion on the left before we can resume focus. This brings us to the left-inversion phase $\Gamma; \Delta \rhd P$, which inverts all of the positive hypotheses on the left starting from the leftmost end of the pattern context, and moves negative hypotheses into Γ. The ordering of Δ forces the left-inversion rules to be applied from left to right, eliminating the irrelevant choice of which order to apply the left-invertible rules from the calculus.

$$\boxed{\Gamma; \Delta \rhd P}$$

$$\frac{\Gamma, N; \Delta \rhd P}{\Gamma; N, \Delta \rhd P}\ \textsc{HypL} \qquad\qquad \frac{\Gamma; \Delta \rhd P}{\Gamma; 1, \Delta \rhd P}\ 1\mathrm{L}$$

$$\frac{\Gamma; A, B, \Delta \rhd P}{\Gamma; A \times B, \Delta \rhd P}\ {\times}\mathrm{L} \qquad\qquad \frac{}{\Gamma; 0, \Delta \rhd P}\ 0\mathrm{L}$$

[1] Intutitionistic products can be either negative, with projective eliminations, or positive, with a let-binding elimination. This corresponds to the fact that intuitionistic products can be seen as corresponding to either one of the linear connectives $A \& B$ or $A \otimes B$. We choose the latter, since it has a let-binding elimination $\mathsf{letpair}\ (x, y) = e\ \mathsf{in}\ e'$ and this is the more

natural elimination to explain pattern matching.

$$\frac{\Gamma; A, \Delta \rhd P \qquad \Gamma; B, \Delta \rhd P}{\Gamma; A + B, \Delta \rhd P}\ +\mathrm{L} \qquad\qquad \frac{\Gamma \vdash P}{\Gamma; \cdot \rhd P}\ \textsc{FocusR}$$

$$\frac{\Gamma \rhd X}{\Gamma; \cdot \rhd X}\ \textsc{FocusL} \qquad\qquad \frac{\Gamma \rhd P \qquad \Gamma; P \rhd Q}{\Gamma; \cdot \rhd Q}\ \textsc{FocusLP}$$

Once the pattern context is emptied, we can resume focus, either returning to right focus via FOCUSR, or by switching into the left focus phase $\Gamma \rhd A$, in which we focus on a hypothesis in the negative context.

$$\boxed{\Gamma \rhd A}$$

$$\frac{A \in \Gamma}{\Gamma \rhd A}\ \textsc{Hyp} \qquad\qquad \frac{\Gamma \rhd A \to B \qquad \Gamma \vdash A}{\Gamma \rhd B}\ {\to}\mathrm{L}$$

Note that this focused sequent calculus is a restriction of the traditional sequent calculus: if we collapsed all of the judgements into a single judgement, elided the focus and blur rules that move us between judgements, and turned the semicolon separating the Γ and Δ contexts into a comma, then each of these rules would be one of the standard rules of the sequent calclus[2]. So it is easily seen that every focused proof has a corresponding proof in the regular sequent calculus. The completeness proof for focusing with respect to the sequent calculus is a little bit harder, and can be found elsewhere (Liang and Miller 2007).

2.2 A Proof Term Assignment

The next step on our path is to assign proof terms to this calculus.

$$
\begin{array}{llcl}
\text{Positive Right} & e & ::= & \langle\rangle \mid \langle e, e' \rangle \mid \mathsf{inl}\ e \mid \mathsf{inr}\ e \mid u \\
\text{Negative Right} & u & ::= & \lambda p.\ u \mid r \\
\text{Arms (Positive Left)} & r & ::= & [] \mid [r \mid r'] \mid e \mid t \mid \mathsf{case}(t, p \Rightarrow r) \\
\text{Applications (Neg. Left)} & t & ::= & x \mid t\ e \\
\text{Patterns} & p & ::= & x \mid \langle\rangle \mid \langle p, p' \rangle \mid [] \mid [p \mid p'] \\
\text{Ordinary Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : N \\
\text{Pattern Contexts} & \Delta & ::= & \cdot \mid \Delta, p : A
\end{array}
$$

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

$$
\frac{}{\Gamma \vdash \langle\rangle : 1}\ 1\text{R}
\qquad\qquad
\frac{\Gamma \vdash e_1 : A \qquad \Gamma \vdash e_2 : B}{\Gamma \vdash \langle e_1, e_2 \rangle : A \times B}\ \times\text{R}
$$

$$
\frac{\Gamma \vdash e : A}{\Gamma \vdash \mathsf{inl}\ e : A + B}\ +\text{R1}
\qquad\qquad
\frac{\Gamma \vdash e : B}{\Gamma \vdash \mathsf{inr}\ e : A + B}\ +\text{R2}
$$

$$
\frac{\Gamma; \cdot \vdash u : N}{\Gamma \vdash u : N}\ \textsc{BlurR}
$$

$$\boxed{\Gamma; \Delta \vdash u : A}$$

$$
\frac{\Gamma; \Delta, p : A \vdash u : B}{\Gamma; \Delta \vdash \lambda p.\ u : A \to B}\ \to\text{R}
\qquad\qquad
\frac{\Gamma; \Delta \rhd r : P}{\Gamma; \Delta \vdash r : P}\ \textsc{BlurL}
$$

$$\boxed{\Gamma; \Delta \rhd r : A}$$

$$\frac{\Gamma, x : N; \Delta \rhd r : P}{\Gamma; x : N, \Delta \rhd r : P}\ \textsc{HypL} \qquad \frac{\Gamma; \Delta \rhd r : P}{\Gamma; \langle\rangle : 1, \Delta \rhd r : P}\ 1\mathrm{L}$$

$$\frac{\Gamma; p : A, p' : B, \Delta \rhd r : P}{\Gamma; \langle p, p' \rangle : A \times B, \Delta \rhd r : P}\ \times\mathrm{L} \qquad \frac{}{\Gamma; [] : 0, \Delta \rhd [] : P}\ 0\mathrm{L}$$

$$\frac{\Gamma; p : A, \Delta \rhd r : P \qquad \Gamma; p' : B, \Delta \rhd r' : P}{\Gamma; [p \mid p'] : A + B, \Delta \rhd [r \mid r'] : P}\ +\mathrm{L}$$

$$\frac{\Gamma \vdash e : P}{\Gamma; \cdot \rhd e : P}\ \textsc{FocusR} \qquad \frac{\Gamma \rhd t : X}{\Gamma; \cdot \rhd t : X}\ \textsc{FocusL}$$

$$\frac{\Gamma \rhd t : P \qquad \Gamma; p : P \rhd r : Q}{\Gamma; \cdot \rhd \mathsf{case}(t, p \Rightarrow r) : Q}\ \textsc{FocusLP}$$

$$\boxed{\Gamma \rhd t : A}$$

$$\frac{x : A \in \Gamma}{\Gamma \rhd x : A}\ \textsc{Hyp} \qquad \frac{\Gamma \rhd t : A \to B \qquad \Gamma \vdash e : A}{\Gamma \rhd t\ e : B}\ \to\mathrm{L}$$

[2] In fact, the →L rule is the natural deduction elimination rule for implication, rather than a true left rule of the sequent calculus. We do this to keep the upcoming proof terms more familiar.

These rules exactly mimic the structure of the sequent calculus we just presented. The proof terms for the right focus judgement $\Gamma \vdash e : A$ consists of all of the introduction forms for the positive types. For the type 1, we have $\langle\rangle$; for products $A \times B$, pairs $\langle e, e' \rangle$; for sums $A + B$, the injections $\mathsf{inl}\ e$ and $\mathsf{inr}\ e$; and the empty type 0 has no introduction form. There is an interesting interaction between the syntax and the typing judgement here; while the negative introductions u are included in the productions of the syntactic class e, the typing rule BLURR only permits us to use these at a negative type. (A similar story can be told for all of the other inclusions — of r in u, and t in r.)

Negative introduction forms — that is, functions — are typed with the judgement $\Gamma; \Delta \vdash u : A$, and the proof terms are lambda-terms $\lambda p.\, u$. Our lambda terms differ in two respects from the usual presentation of functions in the lambda calculus, one minor and one major. The minor difference is that we need no type annotation for the binder — this is because focused calculi are inherently bi-directionally typed.[3] The major difference is that instead of a variable binder $\lambda x.\ u$, we have a *pattern* as the binder for a function. This will be common to all the binding forms of our language, and is where most of the novelty of this calculus resides.

The introduction rule for functions →R moves the pattern from the proof term into the right-hand side of the *pattern context* Δ. The reason we say "right-hand-side" in particular is because the pattern context is an *ordered* context — we cannot make free use of the rule of exchange. So a context $p : A, p' : B, \Delta$ is considered to be different from a context $p' : B, p : A, \Delta$. This is unlike the behavior of the ordinary variable context Γ, in which we identify permutations. Furthermore, instead of mapping variables to types, a pattern context assigns types to patterns — in the previous example p and p' are schematic variables.

The patterns themselves range from the familiar to the unfamiliar. We have a variable pattern x, unit pattern $\langle\rangle$, and pair patterns $\langle p, p'\rangle$, which all match the syntax of patterns from ML. However, we depart quite radically from ML when we reach the case of sum patterns. A pattern for a sum type $A + B$ is a pattern $[p \mid p']$, which can be read as a pattern p for the left branch, and a pattern p' for

STLC	**Focused STLC**
$\mathsf{abort}(t)$	$\mathsf{case}(t, [] \Rightarrow [])$
$\mathsf{case}(t, x_1.\ t_1, x.\ t_2)$	$\mathsf{case}(t, [x_1 \mid x_2] \Rightarrow [t_1 \mid t_2])$
$\mathsf{letunit}\ \langle\rangle = t\ \mathsf{in}\ t'$	$\mathsf{case}(t, \langle\rangle \Rightarrow t')$
$\mathsf{letpair}\ (x, y) = t\ \mathsf{in}\ t'$	$\mathsf{case}(t, \langle x, y\rangle \Rightarrow t')$
$\mathsf{let}\ x = t\ \mathsf{in}\ t'$	$\mathsf{case}(t, x \Rightarrow t')$

Figure 1. Traditional and Focused Eliminations

[3] Concretely, the typing judgements form a well-moded logic program. All of the judgements take the contexts and proof terms as inputs. Bidirectionality arises from the fact that the type argument is an *output* of the $\Gamma \rhd t : A$ judgement, and an input of all of the others.

the right branch. This choice deviates from the ML practice of having separate patterns for each constructor, but it has the benefit of ensuring that a well-typed pattern will always be complete.

After decomposing all of the implications, we have a pattern context Δ and an arm r. Now, we come to the reason why the context must be ordered. Patterns and arms are separate: patterns live in the pattern context on the left of the turnstile $\rhd$, and the body of the proof term, r, carries the arms. So we must regard Δ as an ordered context to track which branches of a sum-pattern $[p \mid p']$ correspond with which arms $[r \mid r']$. This can be seen in the sum elimination rule $+L$ — given a pattern hypothesis $[p \mid p'] : A + B$ at the left-most end of the context, and an arm body $[r \mid r']$, we break it down into two premises, one of which uses the pattern/arm pair $p : A/r$, and the other of which uses $p' : B/r'$. If we were free to permute the pattern hypotheses, then we would lose this correlation.

As an aside, note that functions, as inhabitants of the negative type $A \to B$, are only bound with variable patterns. So the structure of the sequent calculus naturally precludes decomposing functions with pattern matching, showing that this is not an *ad-hoc* restriction. Furthermore, each variable in a pattern becomes a distinct addition to the context Γ, so repeating variables does not require their equality; it will shadow them instead. So the linearity restriction on ML-style patterns can be seen as a restriction that forbids patterns that shadow their own bindings.

After decomposing all of the pattern hypotheses, we can either apply a function with the FOCUSL rule, or we can apply a function and case analyze the result with the FOCUSLP rule. The proof term of the FOCUSLP rule is case analysis, and this offers us an opportunity to examine how the pattern elimination compares with the traditional elimination constructs for sums and products. In Figure 1, we give a table with the traditional eliminator on the left, and the focused elimination on the right. Observe that we have an elimination form for the unit type — this is in analogy to the elimination for the type 1 in linear logic, which is the unit to the tensor $A \otimes B$.

Since the syntax of patterns allows nesting, we can also express case expressions like the following SML expression:

```
case t of
  (Inl x, Inl u) => t1
| (Inr y, Inl u) => t2
| (Inl x, Inr v) => t3
| (Inr y, Inr v) => t4
```

in our focused calculus:

$$\mathsf{case}(t, \langle [x \mid y], [u \mid v] \rangle \Rightarrow [[t_1 \mid t_2] \mid [t_3 \mid t_4]])$$

Instead of writing a series of four disjuncts, we write a single pattern $\langle [x \mid y], [u \mid v] \rangle$, which is a pair pattern with two sum-patterns as subexpressions. The ordering in our context means that the nested arm $[[t_1 \mid t_2] \mid [t_3 \mid t_4]]$ will be decomposed by the left-hand sum-pattern $[x \mid y]$, yielding either $[t_1 \mid t_2]$ or $[t_3 \mid t_4]$, and then the result will be decomposed by the right-hand sum-pattern $[u \mid v]$. In other words, it is also possible to see this pattern calculus as a shorthand for writing nested case statements.

2.3 From Sequent Calculus to Programming Language

All of the well-typed terms of this language are in β-normal, η-long form. This property is enforced by the rules to transition between the judgements, which limit at which types they can be applied. For example, the BLURR rule only allows departing the right positive judgment at a function types, which means that a tuple containing a function component must have it as a lambda-abstraction.

This restriction is extremely useful in a type theory intended for proof search, since this means that the system is cut-free. However, the Curry-Howard correspondence tells us that evaluation of functional programs corresponds to the normalization of proofs. Therefore, we must add the Cut rule back into the calculus in order to get an actual programming language. This introduces non-normal proofs into the calculus, whose corresponding lambda terms are non-normal forms that we can use as programs that actually compute results.

To write η-short terms in our langauge, we relax the restrictions on how long we retain focus and how far we decompose invertible types. We allow hypotheses of arbitrary type in Γ, and relax the BlurR, BlurL, FocusR, FocusL and FocusLP rules so that they accept subterms at any type A, rather than just at positive or negative types. Then, to allow non β-normal terms, we add a type-annotated term $(e : A)$ (with rule ANNOT), which permits normal forms to appear at the heads of case statements and function applications. We also introduce a notion of value for our language suitable for a call-by-value evaluation strategy, with strict sums and products and no evaluation under lambda-abstraction.

$$\begin{array}{llcl}
\text{Negative Elims} & t & ::= & x \mid (e : A) \mid t\ e \\
\text{Values} & v & ::= & \langle\rangle \mid \langle v, v' \rangle \mid \mathsf{inl}\ v \mid \mathsf{inr}\ v \mid \lambda p.\ u
\end{array}$$

$$\Gamma \vdash e : A$$

$$\frac{}{\Gamma \rhd (e : A) : A}\ \textsc{Annot}$$

Now, we need to explain what substitution means for our pattern context. Intuitively, the pattern associated with each hypothesis should drive how values are decomposed, an idea we formalize with the following relations:

$$\boxed{{}^{\mathsf{R}}\langle\!\langle v/p \rangle\!\rangle_A\, u \hookrightarrow u'} \qquad \boxed{{}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A\, r \hookrightarrow r'}$$

$$\frac{{}^{\mathsf{R}}\langle\!\langle v/p \rangle\!\rangle_A\, u \hookrightarrow u'}{{}^{\mathsf{R}}\langle\!\langle v/p \rangle\!\rangle_A\, \lambda p'.\, u \hookrightarrow \lambda p'.\, u'} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A\, r \hookrightarrow r'}{{}^{\mathsf{R}}\langle\!\langle v/p \rangle\!\rangle_A\, r \hookrightarrow r'}$$

$$\frac{}{{}^{\mathsf{L}}\langle\!\langle v/x \rangle\!\rangle_A\, r \hookrightarrow [(v : A)/x]r} \qquad \frac{}{{}^{\mathsf{L}}\langle\!\langle \langle\rangle/\langle\rangle \rangle\!\rangle_1\, r \hookrightarrow r}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v_1/p_1 \rangle\!\rangle_{A_1}\, r \hookrightarrow r' \qquad {}^{\mathsf{L}}\langle\!\langle v_2/p_2 \rangle\!\rangle_{A_2}\, r' \hookrightarrow r''}{{}^{\mathsf{L}}\langle\!\langle \langle v_1, v_2\rangle \,/\, \langle p_1, p_2\rangle \rangle\!\rangle_{A_1 \times A_2}\, r \hookrightarrow r''}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v/p_1 \rangle\!\rangle_A\, r_1 \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle \mathsf{inl}\ v/[p_1 \mid p_2] \rangle\!\rangle_{A+B}\, [r_1 \mid r_2] \hookrightarrow r'}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v/p_2 \rangle\!\rangle_B\, r_2 \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle \mathsf{inr}\ v/[p_1 \mid p_2] \rangle\!\rangle_{A+B}\, [r_1 \mid r_2] \hookrightarrow r'}$$

These rules match a value against a pattern, decomposing the value. When we reach a pair pattern $\langle p_1, p_2\rangle$, we decompose the value $\langle v_1, v_2\rangle$ into two pieces, and sequentially match v_1 against p_1, and then do another pattern substitution on the result, matching v_2 against p_2. When we reach a sum-pattern $[p_1 \mid p_2]$, we decide to take either the left branch or the right branch depending on whether the value is $\mathsf{inl}\ v$ or $\mathsf{inr}\ v$. Finally, when we reach a variable pattern, we substitute the value for the variable. Since variables are in the category of negative focus terms t, we wrap the value in a type annotation $(v : A)$ to ensure that the result is type-correct.

This definition of pattern substitution that satisfies the following principle:

PROPOSITION 1. *If* $\cdot \vdash v : A$*, then:*

- *If* $\Gamma; p : A, \Delta \vdash u : C$*, and* ${}^{\mathsf{R}}\langle\!\langle v/p \rangle\!\rangle_A\, u \hookrightarrow u'$*, then* $\Gamma; \Delta \vdash u' : C$*.*
- *If* $\Gamma; p : A, \Delta \rhd r : C$*, and* ${}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A\, r \hookrightarrow r'$*, then* $\Gamma; \Delta \rhd r' : C$*.*

If we have a value v of the appropriate type, and a well-typed term r in a pattern context with p at the leftmost end, then if the pattern substitution relation relates r to r', we can deduce that r' is well-typed in the smaller pattern context. The following definition of pattern substitution satisfies this principle:

As an aside, it is evident that these rules define a relation that is syntax-directed and total. So it appears we could have defined a pattern substitution function instead of using a more syntactically heavyweight judgement. However, later on we will need to relax this condition, so we present pattern substitution as a relation from the beginning.

Equipped with an understanding of what pattern substitution is, we give the operational semantics below, with mutually recursive reduction relations for each of the syntactic categories. The only novelties in this semantics are that we use pattern substitution instead of ordinary substitution when reducing function applications and case statements, and that there is an extra rule to discard redundant type annotations.

$$\frac{e_1 \mapsto^{\mathsf{PR}} e_1'}{\langle e_1, e_2\rangle \mapsto^{\mathsf{PR}} \langle e_1', e_2\rangle} \qquad \frac{e_2 \mapsto^{\mathsf{PR}} e_2'}{\langle v_1, e_2\rangle \mapsto^{\mathsf{PR}} \langle v_1, e_2'\rangle}$$

$$\frac{e \mapsto^{\mathsf{PR}} e'}{\mathsf{inl}\ e \mapsto^{\mathsf{PR}} \mathsf{inl}\ e'} \qquad \frac{e \mapsto^{\mathsf{PR}} e'}{\mathsf{inr}\ e \mapsto^{\mathsf{PR}} \mathsf{inr}\ e'} \qquad \frac{u \mapsto^{\mathsf{NR}} u'}{u \mapsto^{\mathsf{PR}} u'}$$

$$\frac{r \mapsto^{\mathsf{PL}} r'}{r \mapsto^{\mathsf{NR}} r'} \qquad \frac{t \mapsto^{\mathsf{NL}} t'}{\mathsf{case}(t, p \Rightarrow r) \mapsto^{\mathsf{PL}} \mathsf{case}(t', p \Rightarrow r)}$$

$$\frac{}{(v : A) \mapsto^{\mathsf{PL}} v} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p\rangle\!\rangle_A\, r \hookrightarrow r'}{\mathsf{case}((v : A), p \Rightarrow r) \mapsto^{\mathsf{PL}} r'} \qquad \frac{t \mapsto^{\mathsf{NL}} t'}{t\ e \mapsto^{\mathsf{NL}} t'\ e}$$

$$\frac{e \mapsto^{\mathsf{PR}} e'}{(v : A)\ e \mapsto^{\mathsf{NL}} (v : A)\ e'} \qquad \frac{e \mapsto^{\mathsf{PR}} e'}{(e : A) \mapsto^{\mathsf{NL}} (e' : A)}$$

$$\frac{{}^{\mathsf{R}}\langle\!\langle v/p\rangle\!\rangle_A\, u \hookrightarrow u'}{(\lambda p.\, u : A \to B)\ v \mapsto^{\mathsf{NL}} (u' : B)}$$

This semantics validates the usual progress and preservation theorems.

PROPOSITION 2 (Type Soundness). *This language is sound.*

- *Progress holds:*
  1. *If* $\cdot \vdash e : A$, *then* $e \mapsto^{\mathsf{PR}} e'$ *or* e *is a value* v.
  2. *If* $\cdot ; \cdot \vdash u : A$, *then* $u \mapsto^{\mathsf{NR}} u'$ *or* u *is a value* v.
  3. *If* $\cdot ; \cdot \rhd r : A$, *then* $r \mapsto^{\mathsf{PL}} r'$ *or* r *is a value* v.

4. *If* $\cdot \rhd t : A$*, then* $t \mapsto^{\mathsf{NL}} t'$ *or* t *is a term* $(v : A)$.

- *Type preservation holds:*
  1. *If* $\cdot \vdash e : A$ *and* $e \mapsto^{\mathsf{PR}} e'$*, then* $\cdot \vdash e' : A$
  2. *If* $\cdot;\cdot \vdash u : A$ *and* $u \mapsto^{\mathsf{NR}} u'$*, then* $\cdot;\cdot \vdash u' : A$
  3. *If* $\cdot;\cdot \rhd r : A$ *and* $r \mapsto^{\mathsf{PL}} r'$*, then* $\cdot;\cdot \rhd r' : A$
  4. *If* $\cdot \rhd t : A$ *and* $t \mapsto^{\mathsf{NL}} t'$*, then* $\cdot \vdash t' : A$

3. From ML Patterns to Focused Patterns

While our language is very pleasant theoretically, it is not yet adequate to fully explain ML pattern matching. Consider an example from SML like:

```
case e of
  (Inl x, Inl u)       => e1
| w as (Inr _, Inr _) => e2
| z                    => e3
```

This small example is doing quite a few things. First, it uses wildcard patterns and as-patterns, which have no analog in the pattern language we have described so far. Second, this example relies on implicit priority ordering — we expect the first two patterns to be tested before the last. This is what ensures that even though the variable pattern `z` (in the third clause) matches anything on its own, it will only serve as a catch-all.

To explain as-patterns and wildcards, we will extend the language of patterns with the patterns $\top$ (for wildcard) and the and-pattern $p \wedge p'$. We extend the typing rules and pattern substitution relation below.

$$\frac{\Gamma; \Delta \rhd r : B}{\Gamma; \top : A, \Delta \rhd r : B}\ \textsc{Top} \qquad\qquad \frac{\Gamma; p : A, p' : A, \Delta \rhd r : B}{\Gamma; p \wedge p' : A, \Delta \rhd r : B}\ \textsc{And}$$

$$\frac{}{{}^{\mathsf{L}}\langle\!\langle v/\top \rangle\!\rangle_A\, r \hookrightarrow r} \qquad\qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p_1 \rangle\!\rangle_A\, r \hookrightarrow r' \qquad {}^{\mathsf{L}}\langle\!\langle v/p_2 \rangle\!\rangle_A\, r' \hookrightarrow r''}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \wedge p_2 \rangle\!\rangle_A\, r \hookrightarrow r''}$$

We can introduce a wildcard or and-pattern at any type. The typing rule Top requires that the term be well-typed without the wildcard hypothesis, and the rule And for the and-pattern $p \wedge p'$ requires that we be well-typed in a context with both p and p' pattern hypotheses. If we erase the proof terms from the rules, we see that the rules for these two patterns have a very clear logical interpretation: the rule Top rule is the rule of weakening, and the And rule is the rule of contraction.

The pattern substitution for the $\top$ pattern simply throws away the value and returns the arm unchanged. The and-pattern $p_1 \wedge p_2$ matches v against p_1, and then matches it a second time against p_2.

However, we are no closer to the goal of being able to account for the priority ordering of conventional pattern matching. Looking once more at the example at the start of this section, it is clear that when we interpret the third clause — the pattern $\texttt{z}$ — we must also have some way of saying "match $\texttt{z}$, but not if it matches the first or second clauses". If we had some sort of negation operation on patterns, we could express this constraint, if we interpreted the ML pattern $\texttt{z}$ as the focused pattern $z \wedge \neg[\![\texttt{(Inl x, Inl u)}]\!] \wedge \neg[\![\texttt{(y as (Inr \_, Inr \_))}]\!]$. Here, we use negation to indicate an as-yet-undefined pattern negation, and the semantic brackets to indicate an as-yet-undefined translation of ML patterns into focused form. The idea is that we want a pattern that is z and not the first clause, and not the second clause.

To make this possible, we start with the wildcard $\top$ and and-pattern $p \wedge p'$ patterns, and add their duals to the language of patterns. That is, we add patterns $\bot$ and $p \vee p'$ to our pattern language.

$$\begin{array}{llcllll} \text{Patterns} & p & ::= & \ldots & \mid \; \bot & \mid & p \vee p' \\ \text{Arms} & r & ::= & \ldots & \mid \; \bot & \mid & r \vee r' \end{array}$$

$$\frac{}{\Gamma; \bot : A, \Delta \rhd \bot : B}\ \textsc{Bot}$$

$$\frac{\Gamma; p : A, \Delta \rhd r : B \qquad \Gamma; p' : A, \Delta \rhd r' : B}{\Gamma; p \vee p' : A, \Delta \rhd r \vee r' : B}\ \textsc{Or}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v/p_1 \rangle\!\rangle_A \, r_1 \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \vee p_2 \rangle\!\rangle_A \, r_1 \vee r_2 \hookrightarrow r'} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p_2 \rangle\!\rangle_A \, r_2 \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \vee p_2 \rangle\!\rangle_A \, r_1 \vee r_2 \hookrightarrow r'}$$

The intended semantics of the false-pattern $\bot$ is guaranteed failure — just as $\top$ matches successfully against anything, $\bot$ will successfully match nothing. This semantics is given by *not* giving a rule for $\bot$ in the pattern substitution judgement, which ensures there is no way for a false pattern to match. Likewise, when we match a value against the or-pattern $p \vee p'$, we will nondeterministically choose one or the other pattern to match against. To implement this semantics, we give two rules in the pattern substitution judgement for or-patterns, one corresponding to taking the left branch and one for the right. So pattern substitution may now be undefined (if a false pattern appears), or it may give multiple results (if an or-pattern appears), revealing why we defined it in relational style, as a judgement.

The Or typing rule says that $p \vee p'$ typechecks if the left-hand pattern p typechecks against the left-hand arm r, and the right-hand pattern p' typechecks against the right-hand arm r'. Despite the name, this pattern form is much more general than the or-patterns found in functional programming languages – neither is there a requirement that the two patterns bind the same variables, nor do the two patterns have to share the same arm. Instead, this pattern form is better compared to the vertical bar separating case alternatives $\mathtt{p}_1 \;\rightarrow\; \mathtt{e}_1 \;\mathtt{|}\; \mathtt{p}_2 \;\rightarrow\; \mathtt{e}_2 \;\mathtt{|}\; \ldots \;\mathtt{|}\; \mathtt{p}_n \;\rightarrow\; \mathtt{e}_n$.

The Bot typing rule says that $\bot$ is a valid pattern at all types. It is a syntactic marker for an incomplete pattern match: there is no way for any value to successfully match against it. As a result, the progress lemma fails when the $\bot$ pattern can be used without restriction (though type preservation still holds), because a false-pattern can be used to block reduction in a well-typed program. This is an intentional decision, made for two reasons.

First, internalizing incompleteness as a form of type-unsafety makes it easy to prove the soundness of a coverage algorithm: we know that a coverage algorithm is sound if it is strong enough to make the progress lemma go through. Second, 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. The negation operation has a very similar form to the de Morgan identities:

$$\begin{array}{ll}
\neg x & = \bot \\
\neg \langle\rangle & = \bot \\
\neg \langle p, p' \rangle & = \langle \neg p, \top \rangle \vee \langle \top, \neg p' \rangle \\
\neg [] & = [] \\
\neg [p \mid p'] & = [\neg p \mid \neg p'] \\
\neg \top & = \bot \\
\neg (p \wedge p') & = \neg p \vee \neg p' \\
\neg \bot & = \top \\
\neg (p \vee p') & = \neg p \wedge \neg p'
\end{array}$$

We can show that this definition is a genuine complement.

PROPOSITION 3 (Pattern Negation). *If* $\Gamma; p : A, \Delta \rhd r_1 : C$*,* $\Gamma'; \neg p : A, \Delta' \rhd r_2 : C'$*, and* $\cdot \vdash v : A$*, then:*

- ${}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A\, r_1 \hookrightarrow r'_1$ *or* ${}^{\mathsf{L}}\langle\!\langle v/p' \rangle\!\rangle_A\, r_2 \hookrightarrow r'_2$
- *It is not the case that there exist* r'_1, r'_2 *such that* ${}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A\, r_1 \hookrightarrow r'_1$ *and* ${}^{\mathsf{L}}\langle\!\langle v/\neg p \rangle\!\rangle_A\, r_2 \hookrightarrow r'_2$*.*

So any value v will successfully match against either p or $\neg p$, but not both. The proof of this theorem is a routine induction, but uses the deep operations defined in the next section.

Now, we can explain how to interpret ML-style case statements in terms of focused pattern matching. Suppose we have a case statement with the branches $q_1 \rightarrow e_1 | \ldots q_n \rightarrow e_n$, where we write q for ML pattern expressions:

$$\begin{array}{lccl}
\text{ML Patterns} & q & ::= & x \mid \langle\rangle \mid \langle q, q' \rangle \mid \mathsf{inl}\; q \mid \mathsf{inr}\; q \\
 & & \mid & \_ \mid x\; \mathsf{as}\; q
\end{array}$$

First, we give a simple priority-free interpretation of each pattern, which defines the semantic brackets $[\![q]\!]$ used earlier to motivate our extensions to the pattern language:

$$\begin{array}{lcl}
[\![x]\!] & = & x \\
[\![\langle\rangle]\!] & = & \langle\rangle \\
[\![\langle q_1, q_2\rangle]\!] & = & \langle [\![q_1]\!], [\![q_2]\!]\rangle \\
[\![\mathsf{inl}\ q]\!] & = & [[\![q]\!] \mid \bot] \\
[\![\mathsf{inr}\ q]\!] & = & [\bot \mid [\![q]\!]] \\
[\![\_]\!] & = & \top \\
[\![x\ \mathsf{as}\ q]\!] & = & x \wedge [\![q]\!]
\end{array}$$

Next, we translate a series of disjunctive ML patterns $q_1 \to e_1 | \ldots | q_n \to e_n$ into a pattern/arm pair $(p; r)$ as follows:

```
$\mathsf{translate}(q_1 \to e_1 | \ldots | q_n \to e_n) =$
    $\mathsf{translate}'(q_1 \to e_1 | \ldots | q_n \to e_n; \top)$

$\mathsf{translate}'(\cdot; neg) = (\bot; \bot)$
$\mathsf{translate}'(q_1 \to e_1 | \overrightarrow{q \to e};\ neg) =$
    $\mathsf{let}\ p_1 = [\![q_1]\!] \wedge neg$
    $\mathsf{let}\ r_1 = \mathsf{arm}(p_1; [\![e_1]\!])$
    $\mathsf{let}\ (p; r) = \mathsf{translate}'(\overrightarrow{q \to e};\ \neg[\![q_1]\!] \wedge neg)$
    $(p_1 \vee p; r_1 \vee r)$
```

So q_1 gets translated to $[\![q_1]\!] \wedge \top$, q_2 gets translated to $[\![q_2]\!] \wedge \neg[\![q_1]\!] \wedge \top$, and so on, with all of the negated patterns being stored in the accumulator argument neg. We then disjunctively join the patterns for each branch. Likewise, to translate the ML term e_1 to a pattern term r_1, we first need to find the translation of the term on its own (which we assume to be $[\![e_1]\!]$), and then "expand it" so that the sum- and or-patterns within p_1 can find the term $[\![e_1]\!]$ where they expect to. We give the function to do this below:

$$
\begin{array}{l}
\mathsf{arm}(\cdot; r) = r \\
\mathsf{arm}(x, ps; r) = \mathsf{arm}(ps; r) \\
\mathsf{arm}(\langle\rangle, ps; r) = \mathsf{arm}(ps; r) \\
\mathsf{arm}(\langle p_1, p_2\rangle\ , ps; r) = \mathsf{arm}(p_1, p_2, ps; r) \\
\mathsf{arm}([], ps; r) = [] \\
\mathsf{arm}([p \mid p'], ps; [r \mid r']) = [\mathsf{arm}(p, ps; r) \mid \mathsf{arm}(p', ps; r)] \\
\mathsf{arm}(\top, ps; r) = \mathsf{arm}(ps; r) \\
\mathsf{arm}(p \wedge p', ps; r) = \mathsf{arm}(p, p', ps; r) \\
\mathsf{arm}(\bot, ps; r) = \bot \\
\mathsf{arm}(p \vee p', ps; r) = \mathsf{arm}(p, ps; r) \vee \mathsf{arm}(p', ps; r)
\end{array}
$$

This function walks down the structure of its argument, replicating the or/sum structure of its pattern arguments with or/sum arms, and placing r at the leaves of this tree. (Unsurprisingly, this function exactly mimics the way that the left-inversion phase decomposes the pattern context.)

4. Deep Operations on the Pattern Context

In order to explain features like pattern compilation, we need to be able to manipulate and reorder the pattern context. For example, we would like to be able to simplify a pattern like $[x \mid \bot] \vee [\bot \mid y]$ into a pattern like $[x \mid y]$. However, the pattern context is ordered, in order to fix the order in which case analyses are performed. So theorems about manipulating patterns deep inside the pattern context must also have associated algorithms, which explain how to perform this restructuring.

4.1 Deep Inversion and Deep Introduction Lemmas

We begin by showing that it is admissible to decompose pattern hypotheses anywhere in the context. Each of these lemmas generalizes one of the positive left rules, allowing the decomposition of a pattern hypothesis anywhere within the pattern context. To save space, the lemmas are given in inference rule style, with the double line indicating that the top and the bottom imply each other.

$$\dfrac{\Gamma, x : A; \Delta, \Delta' \rhd r : C}{\Gamma; \Delta, x : A, \Delta' \rhd r : C}\ \textsc{DVar} \qquad \dfrac{\Gamma; \Delta, \Delta' \rhd r : C}{\Gamma; \Delta, \langle\rangle : 1, \Delta' \rhd r : C}\ \mathrm{D}1$$

$$\dfrac{\Gamma; \Delta, p_1 : A_1, p_2 : A_2, \Delta' \rhd r : C}{\Gamma; \Delta, \langle p_1, p_2 \rangle : A_1 \times A_2, \Delta' \rhd r : C}\ \mathrm{D}\times$$

$$\dfrac{\Gamma; \Delta, \Delta' \rhd r : C}{\Gamma; \Delta, \top : A, \Delta' \rhd r : C}\ \mathrm{D}\top$$

$$\dfrac{\Gamma; \Delta, p_1 : A, p_2 : A, \Delta' \rhd r : C}{\Gamma; \Delta, p_1 \wedge p_2 : A, \Delta' \rhd r : C}\ \mathrm{D}\wedge$$

These lemmas show that the inversion and introduction principles for variable, unit, pair, and and-patterns can be generalized to work deep within the pattern context are all admissible, and that doing so does not change the shape of the associated arm.

Next, we generalize the introduction and inversion principles associated with the $+L$ and $\vee L$ rules, and say that it is admissible to apply left-rules for sums and ors deep within the pattern context. However, unlike the case for pair- and and-patterns, doing so *does* require substantially restructuring the arms, because we may need to go deep within the arms to ensure that the case splits in the proof terms still match the order in the context. As a result, we use auxilliary functions to restructure the proof terms appropriately.

$$\dfrac{\begin{array}{c}\Gamma; \Delta, p_1 : A_1, \Delta' \rhd \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r) : C \\ \Gamma; \Delta, p_2 : A_2, \Delta' \rhd \mathsf{Out}^{\mathsf{R}}_{+}(\Delta; r) : C\end{array}}{\Gamma; \Delta, [p_1 \mid p_2] : A_1 + A_2, \Delta' \rhd r : C}\ \mathrm{D}{+}\textsc{Out}$$

$$\Gamma; \Delta, p_1 : A, \Delta' \rhd \mathsf{Out}^{\mathsf{L}}_{\vee}(\Delta; r) : C$$

$$\frac{\Gamma; \Delta, p_2 : A, \Delta' \rhd \mathsf{Out}^{\mathsf{R}}_{\vee}(\Delta; r) : C}{\Gamma; \Delta, p_1 \vee p_2 : A, \Delta' \rhd r : C}\ \textsc{D}\vee\textsc{Out}$$

$$\frac{\Gamma; \Delta, p_1 : A_1, \Delta' \rhd r_1 : C \qquad \Gamma; \Delta, p_2 : A_2, \Delta' \rhd r_2 : C}{\Gamma; \Delta, [p_1 \mid p_2] : A_1 + A_2, \Delta' \rhd \mathsf{Join}_{+}(\Delta; r_1; r_2) : C}\ \textsc{D+Join}$$

$$\frac{\Gamma; \Delta, p_1 : A, \Delta' \rhd r_1 : C \qquad \Gamma; \Delta, p_2 : A, \Delta' \rhd r_2 : C}{\Gamma; \Delta, p_1 \vee p_2 : A, \Delta' \rhd \mathsf{Join}_{\vee}(\Delta; r_1; r_2) : C}\ \textsc{D}\vee\textsc{Join}$$

In the D$\vee$JOIN and D+JOIN rules, we use the functions $\mathsf{Join}_{\vee}$ and Join_{+} to properly combine the two arms r_1 and r_2. Here is the definition of the Join_{+} function:

$$\begin{array}{ll}
\mathsf{Join}_{+}(\cdot; r_1; r_2) & = [r_1 \mid r_2] \\
\mathsf{Join}_{+}(x : A, \Delta; r_1; r_2) & = \mathsf{Join}_{+}(\Delta; r_1; r_2) \\
\mathsf{Join}_{+}(\langle\rangle : 1, \Delta; r_1; r_2) & = \mathsf{Join}_{+}(\Delta; r_1; r_2) \\
\mathsf{Join}_{+}(\langle p_1, p_2 \rangle : A_1 \times A_2, \Delta; r_1; r_2) & = \mathsf{Join}_{+}(p_1 : A_1, p_2 : A_2, \Delta; r_1; r_2) \\
\mathsf{Join}_{+}([] : 0, \Delta; r_1; r_2) & = [] \\
\multicolumn{2}{l}{\mathsf{Join}_{+}([p_1 \mid p_2] : A + B, \Delta; [r_1' \mid r_1'']; [r_2' \mid r_2'']) =} \\
\multicolumn{2}{l}{\quad [\mathsf{Join}_{+}(p_1 : A, \Delta; r_1'; r_2') \mid \mathsf{Join}_{+}(p_2 : B, \Delta; r_1''; r_2'')]} \\
\mathsf{Join}_{+}(\top : A, \Delta; r_1; r_2) & = \mathsf{Join}_{+}(\Delta; r_1; r_2) \\
\mathsf{Join}_{+}(p_1 \wedge p_2 : A, \Delta; r_1; r_2) & = \mathsf{Join}_{+}(p_1 : A, p_2 : A, \Delta; r_1; r_2) \\
\mathsf{Join}_{+}(\bot : A, \Delta; r_1; r_2) & = \bot \\
\multicolumn{2}{l}{\mathsf{Join}_{+}(p_1 \vee p_2 : A, \Delta; r_1' \vee r_1''; r_2' \vee r_2'') =} \\
\multicolumn{2}{l}{\quad \mathsf{Join}_{+}(p_1 : A, \Delta; r_1'; r_2') \vee \mathsf{Join}_{+}(p_2 : a, \Delta; r_1''; r_2'')}
\end{array}$$

This function walks down the structure of the pattern contexts, destructuring the arms in lockstep (and hence their proof trees), until it reaches the leaves of the derivation, where the goal patterns are at the leftmost position. Then it simply puts the two derivations together according to the $+$L rule. Note that this function definition is apparently partial — when the head of the context argument Δ is a sum pattern $[p \mid p']$ or an or-pattern $p \vee p'$, then both of the term arguments must be either sum-case bodies $[r \mid r']$ or or-pattern bodies $r \vee r'$ respectively. However, well-typed terms always satisfy this condition. (The definition of $\mathsf{Join}_\vee$ is similar; the only difference is that the base case becomes $\mathsf{Join}_\vee(\cdot; r_1; r_2) = r_1 \vee r_2$.)

Likewise, the D+OUT rule shows that we can take apart a term based on a case analysis it performs deep within its arm. The OutL function walks down the structure of the context to grab the correct branch of the case at each leaf:

$$\begin{array}{ll}
\mathsf{Out}^{\mathsf{L}}_{+}(\cdot; [r_1 \mid r_2]) & = r_1 \\
\mathsf{Out}^{\mathsf{L}}_{+}(x : A, \Delta; r) & = \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r) \\
\mathsf{Out}^{\mathsf{L}}_{+}(\langle\rangle : 1, \Delta; r) & = \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r) \\
\mathsf{Out}^{\mathsf{L}}_{+}(\langle p_1, p_2\rangle : A_1 \times A_2, \Delta; r) & = \mathsf{Out}^{\mathsf{L}}_{+}(p_1 : A_1, p_2 : A_2, \Delta; r) \\
\mathsf{Out}^{\mathsf{L}}_{+}([] : 0, \Delta; r) & = [] \\
\multicolumn{2}{l}{\mathsf{Out}^{\mathsf{L}}_{+}([p_1 \mid p_2] : A_1 + A_2, \Delta; [r_1 \mid r_2]) =} \\
\multicolumn{2}{l}{\quad [\mathsf{Out}^{\mathsf{L}}_{+}(p_1 : A_1, \Delta; r_1) \mid \mathsf{Out}^{\mathsf{L}}_{+}(p_2 : A_2, \Delta; r_2)]} \\
\mathsf{Out}^{\mathsf{L}}_{+}(\top : A, \Delta; r) & = \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r) \\
\mathsf{Out}^{\mathsf{L}}_{+}(p_1 \wedge p_2 : A, \Delta; r) & = \mathsf{Out}^{\mathsf{L}}_{+}(p_1 : A, p_2 : A, \Delta; r) \\
\mathsf{Out}^{\mathsf{L}}_{+}(\bot : A, \Delta; r) & = \bot \\
\multicolumn{2}{l}{\mathsf{Out}^{\mathsf{L}}_{+}(p_1 \vee p_2 : A, \Delta; r_1 \vee r_2) =} \\
\multicolumn{2}{l}{\quad \mathsf{Out}^{\mathsf{L}}_{+}(p_1 : A, \Delta; r_1) \vee \mathsf{Out}^{\mathsf{L}}_{+}(p_2 : A, \Delta; r_2)}
\end{array}$$

The other Out projections are identical, except with differing base cases as follows:

$$\begin{array}{l}
\mathsf{Out}^{\mathsf{R}}_{+}(\cdot; [r_1 \mid r_2]) = r_2 \\
\mathsf{Out}^{\mathsf{L}}_{\vee}(\cdot; r_1 \vee r_2) \ = r_1 \\
\mathsf{Out}^{\mathsf{R}}_{\vee}(\cdot; r_1 \vee r_2) \ = r_2
\end{array}$$

Finally, we have generalized rules for the 0 type and false-pattern.

$$\frac{}{\Gamma; \Delta, [] : 0, \Delta' \rhd \mathsf{Abort}_0(\Delta) : C}\ \mathrm{D0}$$

$$\frac{}{\Gamma; \Delta, \bot : A, \Delta' \rhd \mathsf{Abort}_{\bot}(\Delta) : C}\ \mathrm{D}\bot$$

As expected, there are associated functions to ensure that an abort or a $\bot$ occurs at every leaf of the proof tree:

$$\begin{array}{ll}
\mathsf{Abort}_0(\cdot) & = [] \\
\mathsf{Abort}_0(x : A, \Delta) & = \mathsf{Abort}_0(\Delta) \\
\mathsf{Abort}_0(\langle\rangle : 1, \Delta) & = \mathsf{Abort}_0(\Delta) \\
\mathsf{Abort}_0(\langle p_1, p_2\rangle : A_1 \times A_2, \Delta) & = \mathsf{Abort}_0(p_1 : A_1, p_2 : A_2, \Delta) \\
\mathsf{Abort}_0([] : 0, \Delta) & = [] \\
\mathsf{Abort}_0([p_1 \mid p_2] : A_1 + A_2, \Delta) & = \\
\quad [\mathsf{Abort}_0(p_1 : A_1, \Delta) \mid \mathsf{Abort}_0(p_2 : A_2, \Delta)] & \\
\mathsf{Abort}_0(\top : A, \Delta) & = \mathsf{Abort}_0(\Delta) \\
\mathsf{Abort}_0(p_1 \wedge p_2 : A, \Delta) & = \mathsf{Abort}_0(p_1 : A, p_2 : A, \Delta) \\
\mathsf{Abort}_0(\bot : A, \Delta) & = \bot \\
\mathsf{Abort}_0(p_1 \vee p_2 : A, \Delta) & = \\
\quad \mathsf{Abort}_0(p_1 : A, \Delta) \vee \mathsf{Abort}_0(p_2 : A, \Delta) &
\end{array}$$

$\mathsf{Abort}_{\bot}(\Delta)$ is similar, with the first case returning $\bot$.

4.2 Coherence of the Deep Rules

The deep inversions let us destructure terms based on a pattern hypothesis anywhere in the pattern context, and the deep introductions let us re-construct them. Beyond the admissibility of the deep inversion and introduction rules, we will need the following coherence properties:

PROPOSITION 4. *The following equations hold for all Δ, r, r_1, r_2*

- $\mathsf{Join}_{\vee}(\Delta; r_1; r_2) = r$ *if and only if* $r_1 = \mathsf{Out}^{\mathsf{L}}_{\vee}(\Delta; r)$ *and* $r_2 = \mathsf{Out}^{\mathsf{R}}_{\vee}(\Delta; r)$*,*
- $\mathsf{Join}_{+}(\Delta; r_1; r_2) = r$ *if and only if* $r_1 = \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r)$ *and* $r_2 = \mathsf{Out}^{\mathsf{R}}_{+}(\Delta; r)$*,*

- *If* $\Gamma; \Delta, [] : 0, \Delta' \rhd r : C$*, then* $r = \mathsf{Abort}_0(\Delta)$,
- *If* $\Gamma; \Delta, \bot : A, \Delta' \rhd r : C$*, then* $r = \mathsf{Abort}_\bot(\Delta)$.

These properties give us a "round-trip" property — if we take a term apart with a deep inversion and put it back together with a deep introduction (or vice-versa), then we get back the term we started with. In addition to this, we will need some lemmas that let enable us to commute uses of Join and Out.

PROPOSITION 5. *For all* $\oplus \in \{+, \vee\}$ *and* $\mathsf{d} \in \{\mathsf{L}, \mathsf{R}\}$*, we have:*

- $r' = \mathsf{Out}^{\mathsf{d}}_{\vee}(\Delta; \mathsf{Out}^{\mathsf{d}'}_{\oplus}(\Delta, p_{\mathsf{L}} \vee p_{\mathsf{R}} : A, \Delta'; r))$ *iff* $r' = \mathsf{Out}^{\mathsf{d}'}_{\oplus}(\Delta, p_{\mathsf{d}} : A, \Delta'; \mathsf{Out}^{\mathsf{d}}_{\vee}(\Delta; r))$.
- $r' = \mathsf{Out}^{\mathsf{d}}_{\oplus}(\Delta, p_{\mathsf{L}} \vee p_{\mathsf{R}} : A, \Delta'; \mathsf{Join}_{\vee}(\Delta; r_1; r_2))$ *iff* $r' = \mathsf{Join}_{\vee}(\Delta; \mathsf{Out}^{\mathsf{d}}_{\oplus}(\Delta, p_{\mathsf{L}} : A, \Delta'; r_1); \mathsf{Out}^{\mathsf{d}}_{\oplus}(\Delta, p_{\mathsf{R}} : A, \Delta'; r_2))$
- $r' = \mathsf{Out}^{\mathsf{d}}_{\vee}(\Delta; \mathsf{Join}_{\oplus}(\Delta, p_{\mathsf{L}} \vee p_{\mathsf{R}} : A, \Delta'; r_1; r_2))$ *iff* $r' = \mathsf{Join}_{\oplus}(\Delta, p_{\mathsf{d}} : A, \Delta'; \mathsf{Out}^{\mathsf{d}}_{\vee}(\Delta; r_1); \mathsf{Out}^{\mathsf{d}}_{\vee}(\Delta; r_2))$
- $r' = \mathsf{Join}_{\oplus}(\Delta, p_{\mathsf{L}} \vee p_{\mathsf{R}} : A, \Delta'; \mathsf{Join}_{\vee}(\Delta; r_1; r_1'); \mathsf{Join}_{\vee}(\Delta; r_2; r_2'))$ *iff* $r' =$ $\mathsf{Join}_{\vee}(\Delta; \mathsf{Join}_{\oplus}(\Delta, p_{\mathsf{L}} : A, \Delta'; r_1; r_2); \mathsf{Join}_{\oplus}(\Delta, p_{\mathsf{R}} : A, \Delta'; r_1'; r_2'))$

In addition to these equalities, we have another four similar cases where the $\vee$-functions are replaced with $+$-functions. The net effect of these equalities is that we can use the deep inversions and introductions in any order, with the confidence that the order will not change the final result term — we get the same result term regardless of the path to its construction.

4.3 Deep Exchange and Substitution

Since reordering the pattern context corresponds to doing case splits in different orders, it seems intuitively clear that reordering the pattern context should give rise to a new proof term that is in some sense equivalent to the old one. To make this idea precise, we do three things. First, we prove that the rule of *exchange* is admissible for the pattern context. Second, we extend the notion of pattern substitution to include substitutions deep within the context, and not just at the leftmost hypothesis. Finally, we show that if we get the same result regardless of whether deep substitution follows an exchange, or preceeds it, which justifies the intuition that moving the pattern hypothesis in the ordered context "didn't matter".

4.3.1 Exchange

PROPOSITION 6 (Exchange). *If* $\Gamma; \Delta, p : A, \Delta', \Delta'' \rhd r : C$ *holds, then we have* $\Gamma; \Delta, \Delta', p : A, \Delta'' \rhd \mathsf{Ex}(\Delta; p : A; \Delta'; r) : C$.

Here, we assert that if we have a derivation of $\Gamma; \Delta, p : A, \Delta', \Delta'' \rhd r : C$, then we can move the hypothesis $p : A$ to the right, past the assumptions in Δ'. The proof term naturally gets altered, and so we must give a function Ex which computes the new arm.

```
$\mathsf{Ex}(\Delta; x : A; \Delta'; r) = r$
$\mathsf{Ex}(\Delta; \langle\rangle : 1; \Delta'; r) = r$
$\mathsf{Ex}(\Delta; \langle p_1, p_2\rangle : A_1 \times A_2; \Delta'; r) =$
    $\mathsf{let}\ r' = \mathsf{Ex}(\Delta; p_1 : A_1; p_2 : A_2, \Delta'; r)$
    $\mathsf{Ex}(\Delta; p_2 : A_2; \Delta', p_1 : A_1; r')$
$\mathsf{Ex}(\Delta; [] : 0; \Delta'; r) = \mathsf{Abort}_0(\Delta)$
$\mathsf{Ex}(\Delta; [p_1 \mid p_2] : A_1 + A_2; \Delta'; r) =$
    $\mathsf{let}\ r'_1 = \mathsf{Ex}(\Delta; p_1 : A_1; \Delta'; \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r))$
    $\mathsf{let}\ r'_2 = \mathsf{Ex}(\Delta; p_2 : A_2; \Delta'; \mathsf{Out}^{\mathsf{R}}_{+}(\Delta; r))$
    $\mathsf{Join}_{+}(\Delta, \Delta'; r'_1; r'_2)$
$\mathsf{Ex}(\Delta; \top : A; \Delta'; r) = r$
$\mathsf{Ex}(\Delta; p_1 \wedge p_2 : A; \Delta'; r) =$
    $\mathsf{let}\ r' = \mathsf{Ex}(\Delta; p_1 : A; p_2 : A, \Delta'; r)$
    $\mathsf{Ex}(\Delta; p_2 : A; \Delta', p_1 : A_1; r')$
```

$$\begin{array}{l}
\mathsf{Ex}(\Delta; \bot : A; \Delta'; r) = \mathsf{Abort}_{\bot}(\Delta) \\
\mathsf{Ex}(\Delta; p_1 \vee p_2 : A_1 + A_2; \Delta'; r) = \\
\qquad \mathsf{let}\ r'_1 = \mathsf{Ex}(\Delta; p_1 : A_1; \Delta'; \mathsf{Out}^{\mathsf{L}}_{\vee}(\Delta; r)) \\
\qquad \mathsf{let}\ r'_2 = \mathsf{Ex}(\Delta; p_2 : A_2; \Delta'; \mathsf{Out}^{\mathsf{R}}_{\vee}(\Delta; r)) \\
\qquad \mathsf{Join}_{\vee}(\Delta, \Delta'; r'_1; r'_2)
\end{array}$$

This function is inductively defined on the structure of the pattern argument p, and it uses the deep inversion principles (and their associated functions Out) to break down the derivation for the recursive calls, and it uses the deep introduction functions (and their associated functions Join) to rebuild the arm for the new pattern context.

4.3.2 Deep Substitution

The statement of the deep pattern substitution lemma is a straightforward generalization of the substitution principle for the pattern context:

PROPOSITION 7. *If* $\cdot \vdash v : A$, *and* $\Gamma; \Delta, p : A, \Delta' \rhd r : C$, *and* ${}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r'$, *then* $\Gamma; \Delta, \Delta' \rhd r' : C$.

Of course, we need to define deep pattern substitution. In the definition below, note that it is nearly identical to the ordinary pattern substitution — we simply index the relation by Δ, and use the Join_* and Out^*_* functions in the place of the $[\cdot \mid \cdot]$ and $\cdot \vee \cdot$ constructors.

$$\dfrac{}{{}^{\mathsf{L}}\langle\!\langle v/x \rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow [(v : A)/x]r} \qquad \dfrac{}{{}^{\mathsf{L}}\langle\!\langle \langle\rangle/\langle\rangle \rangle\!\rangle^{\Delta}_{1}\, r \hookrightarrow r}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v_1/p_1\rangle\!\rangle^{\Delta}_{A_1}\, r \hookrightarrow r' \qquad {}^{\mathsf{L}}\langle\!\langle v_2/p_2\rangle\!\rangle^{\Delta}_{A_2}\, r' \hookrightarrow r''}{{}^{\mathsf{L}}\langle\!\langle \langle v_1, v_2\rangle \;/\; \langle p_1, p_2\rangle\rangle\!\rangle^{\Delta}_{A_1 \times A_2}\, r \hookrightarrow r''}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v/p_1\rangle\!\rangle^{\Delta}_{A}\, (\mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r)) \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle \mathsf{inl}\ v/[p_1 \mid p_2]\rangle\!\rangle^{\Delta}_{A+B}\, r \hookrightarrow r'} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p_2\rangle\!\rangle^{\Delta}_{A}\, (\mathsf{Out}^{\mathsf{R}}_{+}(\Delta; r)) \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle \mathsf{inr}\ v/[p_1 \mid p_2]\rangle\!\rangle^{\Delta}_{A+B}\, r \hookrightarrow r'}$$

$$\frac{}{{}^{\mathsf{L}}\langle\!\langle v/\top\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p_1\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r' \qquad {}^{\mathsf{L}}\langle\!\langle v/p_2\rangle\!\rangle^{\Delta}_{A}\, r' \hookrightarrow r''}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \wedge p_2\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r''}$$

$$\frac{{}^{\mathsf{L}}\langle\!\langle v/p_1\rangle\!\rangle^{\Delta}_{A}\, (\mathsf{Out}^{\mathsf{L}}_{\vee}(\Delta; r)) \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \vee p_2\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r'} \qquad \frac{{}^{\mathsf{L}}\langle\!\langle v/p_2\rangle\!\rangle^{\Delta}_{A}\, (\mathsf{Out}^{\mathsf{R}}_{\vee}(\Delta; r)) \hookrightarrow r'}{{}^{\mathsf{L}}\langle\!\langle v/p_1 \vee p_2\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r'}$$

4.3.3 Permuting Substitution and Exchange

Now, we can show that we can permute the order in which we do substitutions and exchanges. That is, we want to show that if we substitute a value for a pattern in a context after doing an exchange, we get the same result as performing the exchange, and then substituting into the permuted context. We end up with four cases to this lemma, depending on whether the target hypothesis of the exchange is to the right of the hypothesis to be substituted, the pattern to be substituted itself, or is a hypothesis to the left of the substitution target. For clarity, we write these lemmas in inference rule style, putting the premises above the line and the conclusion below. In the following two cases, we assume $\cdot \vdash v : A$:

$${}^{\mathsf{L}}\langle\!\langle v/p\rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r' \qquad \Gamma; \Delta, p : A, \Delta', p' : B, \Delta'' \rhd r : C$$

$$\frac{}{\mathsf{L}\langle\!\langle v/p\rangle\!\rangle^{\Delta}_{A}\ \mathsf{Ex}(\Delta, p : A, \Delta'; p' : B; \Delta''; r) \hookrightarrow \mathsf{Ex}(\Delta, \Delta'; p' : B; \Delta''; r')}$$

$$\frac{\mathsf{L}\langle\!\langle v/p\rangle\!\rangle^{\Delta}_{A}\ r \hookrightarrow r' \qquad \Gamma; \Delta, p : A, \Delta', \Delta'' \rhd r : C}{\mathsf{L}\langle\!\langle v/p\rangle\!\rangle^{\Delta,\Delta'}_{A}\ \mathsf{Ex}(\Delta; p : A; \Delta'; r) \hookrightarrow r'}$$

In the first case, we exchange the position of a hypothesis to the right of the substitution target, and in the conclusion we see that performing the substitution on the exchanged term yields the same result as performing exchange on the substitutand. In the second case, we see that exchanging the target of a substitution does not change the result.

In the next two cases, we assume that $\cdot \vdash v : B$:

$$\frac{\mathsf{L}\langle\!\langle v/p'\rangle\!\rangle^{\Delta,p:A,\Delta',\Delta''}_{B}\ r \hookrightarrow r' \qquad \Gamma; \Delta, p : A, \Delta', \Delta'', p' : B, \Delta''' \rhd r : C}{\mathsf{L}\langle\!\langle v/p'\rangle\!\rangle^{\Delta,\Delta',p:A,\Delta''}_{B}\ \mathsf{Ex}(\Delta; p : A; \Delta'; r) \hookrightarrow \mathsf{Ex}(\Delta; p : A; \Delta'; r')}$$

$$\frac{\mathsf{L}\langle\!\langle v/p'\rangle\!\rangle^{\Delta,p:A,\Delta'}_{B}\ r \hookrightarrow r' \qquad \Gamma; \Delta, p : A, \Delta', p' : B, \Delta'', \Delta''' \rhd r : C}{\mathsf{L}\langle\!\langle v/p'\rangle\!\rangle^{\Delta,\Delta'}_{B}\ \mathsf{Ex}(\Delta; p : A; \Delta', p' : B, \Delta''; r) \hookrightarrow \mathsf{Ex}(\Delta; p : A; \Delta', \Delta''; r')}$$

In the third case, the target of the exchange is to the left of the substitution target, but is not exchanged past it. In the fourth case, the exchange target is to the left of the substitution target, and the action of the exchange moves it to the right of the substitution target.

Proving this commutation involves proving it for Out and Join, and then working up.

4.3.4 Discussion

In this section, we have stated numerous technical lemmas. The reason for going into this level of detail is to illustrate that the overall structure of the metatheorems 1) is extremely regular and 2) consists of statements of familiar logical principles. The only thing that makes these lemmas differ from the corresponding lemmas for the sequent calculus formulation is that we have to explicitly manage the proof terms. Even so, we can see the algorithms as nothing more than the constructive content of the proofs showing the admissibility of principles like inversion, introduction, exchange and substitution. When we use these algorithms as subroutines to implement coverage checking and pattern compilation, we will be able to see exactly how directly they depend on logical proof transformations.

5. Coverage Checking

In order to get useful answers out of the machinery of the previous sections, we first need some good questions. One question is the question of coverage checking — how can we ensure that a pattern match against a value will always yield a unique result?

This is an interesting question because the pattern language we have introduced does not always answer these questions in the affirmative. For example, the pattern $[x \mid \bot] \wedge [\bot \mid y]$ will never match against any value, and likewise, the pattern $x \vee \langle \top, y \rangle$ can match in two different ways.

So we must find out how to check whether the pattern substitution relation defines a total function for a particular pattern. Since a relation is functional when it is total and it has only one output for each input, we will define judgements to track both of these conditions. We begin with the judgement $p \;\mathsf{det}\; A$, which should hold whenever there is at most one way any value of type A can match against the pattern p.

$$\boxed{p \;\mathsf{det}\; A}$$

$$\frac{}{x\ \mathsf{det}\ A} \qquad \frac{}{[]\ \mathsf{det}\ 0} \qquad \frac{p_1\ \mathsf{det}\ A_1 \qquad p_2\ \mathsf{det}\ A_2}{[p_1 \mid p_2]\ \mathsf{det}\ A_1 + A_2} \qquad \frac{}{\langle\rangle\ \mathsf{det}\ 1}$$

$$\frac{p_1\ \mathsf{det}\ A_1 \qquad p_2\ \mathsf{det}\ A_2}{\langle p_1, p_2\rangle\ \mathsf{det}\ A_1 \times A_2} \qquad \frac{}{\top\ \mathsf{det}\ A} \qquad \frac{p_1\ \mathsf{det}\ A \qquad p_2\ \mathsf{det}\ A}{p_1 \wedge p_2\ \mathsf{det}\ A}$$

$$\frac{}{\bot\ \mathsf{det}\ A} \qquad \frac{p_1\ \mathsf{det}\ A \qquad p_2\ \mathsf{det}\ A \qquad p_1, p_2\ \mathsf{fail}\ A}{p_1 \vee p_2\ \mathsf{det}\ A}$$

This judgement inductively follows the structure of the pattern, asking only that each pattern's sub-components are deterministic, until it reaches the case for the or-pattern $p \vee p'$. This is the difficult case, because both p and p' might be deterministic, but the combination might not be. For example, consider the pattern $x \vee \top$. Each half is trivially deterministic, but because there are values that can match either pattern, the or-pattern as a whole is not determinate.

If we knew that there were no values that could match both patterns, then we could know the whole pattern covers. To determine this, we introduce the judgement $p_1, \ldots, p_n\ \mathsf{fail}\ A$, which holds when no value v of type A can match against all of the patterns p_i. Now, we call out to $p_1, p_2\ \mathsf{fail}\ A$ to establish the premise of the or-pattern case — we believe $p_1 \vee p_2\ \mathsf{det}\ A$ when we know that there is nothing in the intersection of p_1 and p_2. We give the definition of $\overrightarrow{p}\ \mathsf{fail}\ A$ itself below:

$$\boxed{\overrightarrow{p}\ \mathsf{fail}\ A}$$

$$\overrightarrow{p_1}, \overrightarrow{p_2}\ \mathsf{fail}\ A \qquad \qquad \overrightarrow{p_1}, p, p', \overrightarrow{p_2}\ \mathsf{fail}\ A$$

$$\frac{}{\overrightarrow{p_1}, \top, \overrightarrow{p_2}\ \mathsf{fail}\ A} \qquad \frac{}{\overrightarrow{p_1}, p \wedge p', \overrightarrow{p_2}\ \mathsf{fail}\ A} \qquad \frac{}{\overrightarrow{p_1}, \bot, \overrightarrow{p_2}\ \mathsf{fail}\ A}$$

$$\frac{\overrightarrow{p_1}, p, \overrightarrow{p_2}\ \mathsf{fail}\ A \qquad \overrightarrow{p_1}, p', \overrightarrow{p_2}\ \mathsf{fail}\ A}{\overrightarrow{p_1}, p \vee p', \overrightarrow{p_2}\ \mathsf{fail}\ A}$$

$$\frac{\overrightarrow{p_1}, \overrightarrow{p_2}\ \mathsf{fail}\ A}{\overrightarrow{p_1}, x, \overrightarrow{p_2}\ \mathsf{fail}\ A} \qquad \frac{}{\overrightarrow{[]}\ \mathsf{fail}\ 0} \qquad \frac{\overrightarrow{p_1}\ \mathsf{fail}\ A_1 \qquad \overrightarrow{p_2}\ \mathsf{fail}\ A_2}{\overrightarrow{[p_1 \mid p_2]}\ \mathsf{fail}\ A_1 + A_2}$$

$$\frac{\overrightarrow{p_1}\ \mathsf{fail}\ A_1}{\overrightarrow{\langle p_1, p_2 \rangle}\ \mathsf{fail}\ A_1 \times A_2} \qquad \frac{\overrightarrow{p_2}\ \mathsf{fail}\ A_2}{\overrightarrow{\langle p_1, p_2 \rangle}\ \mathsf{fail}\ A_1 \times A_2}$$

Furthermore, we can use the fact that we have a syntactic negation on patterns to turn our failure judgement into a coverage judgement. That is, if $\neg p\ \mathsf{fail}\ A$, then we know that p must match against all values of type A — which is precisely what we require of coverage!

$$\frac{(\neg p)\ \mathsf{fail}\ A}{p\ \mathsf{covers}\ A}$$

Now, we can make these informal claims precise, and show the soundness of these judgements.

PROPOSITION 8 (Failure). *If we have derivations*

- $p_1, \ldots, p_n\ \mathsf{fail}\ A$
- $\Gamma; \Delta, p_1 : A, \ldots, p_n : A, \Delta' \rhd r_1 : C,$
- $\cdot \vdash v : A$

then it is not the case that there exist $r_2, \ldots, r_{n+1}$ *such that for all* $i \in \{1, \ldots, n\}$, ${}^{\mathsf{L}}\langle\!\langle v/p_i \rangle\!\rangle^{\Delta}_{A}\, r_i \hookrightarrow r_{i+1}$.

The proof of this statement is via an induction on the failure judgement, and in each case we make use of the deep operations defined in the previous section.

The most interesting case of the proof is when we reach the rule for decomposing a set of pair patterns $\overrightarrow{\langle p_1, p_2 \rangle}$. In this case, we need to make use of the lemmas that allow us to commute exchange and substitution — we begin with a context in the form $\Delta, \overrightarrow{\langle p_1, p_2 \rangle : A_1 \times A_2}, \Delta'$. Inversion lets us rewrite the context to $\Delta, \overrightarrow{p_1 : A_1, p_2 : A_2}, \Delta'$. Then, we use exchange to transform it into the form $\Delta, \overrightarrow{p_1 : A_1}, \overrightarrow{p_2 : A_2}, \Delta'$, and appeal to the induction hypothesis. This gives us enough evidence to construct the refutation we need only because we can reorder a series of exchanges and substitutions without changing the final result.

PROPOSITION 9 (Coverage). *If we have derivations* $\cdot \vdash v : A$, p covers A, *and* $\Gamma; \Delta, p : A, \Delta' \rhd r : C$, *then* ${}^{\llcorner}\langle\langle v/p \rangle\rangle^{\Delta}_{A}\, r \hookrightarrow r'$.

This is an immediate consequence of the soundness of the failure judgement, and the semantics of pattern negation.

PROPOSITION 10 (Determinacy). *If we have derivations*

- $\cdot \vdash v : A$,
- p det A,
- $\Gamma; \Delta, p : A, \Delta' \rhd r : C$,
- $D_1 :: {}^{\llcorner}\langle\langle v/p \rangle\rangle^{\Delta}_{A}\, r \hookrightarrow r'$,

then if $D_2 :: {}^{\llcorner}\langle\langle v/p \rangle\rangle^{\Delta}_{A}\, r \hookrightarrow r''$, *we know* $D_1 = D_2$.

We write $D :: \ {}^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A^\Delta\, r \hookrightarrow r'$ to indicate that we want to consider a *particular* derivation D of the pattern substitution of v into p. This means that our determinacy judgement ensures that there is at most one way to perform a pattern substitution for each value.

We can now recover the progress lemma, by changing the rules governing the introduction of pattern hypotheses:

$$\frac{\Gamma; \Delta, p : A \vdash u : B \qquad p \ \textsf{covers}\ A \qquad p \ \textsf{det}\ A}{\Gamma; \Delta \vdash \lambda p.\, u : A \to B} \to\text{R'}$$

$$\frac{\Gamma \rhd t : A \qquad \Gamma; p : A \rhd r : B \qquad p \ \textsf{covers}\ A \qquad p \ \textsf{det}\ A}{\Gamma; \cdot \rhd \textsf{case}(t, p \Rightarrow r) : B} \ \text{FocusP'}$$

PROPOSITION 11 (Progress, Redux). *We have that:*

1. *If* $\cdot \vdash e : A$*, then* $e \mapsto^{\mathsf{PR}} e'$ *or* e *is a value* v*.*
2. *If* $\cdot; \cdot \vdash u : A$*, then* $u \mapsto^{\mathsf{NR}} u'$ *or* u *is a value* v*.*
3. *If* $\cdot; \cdot \rhd r : A$*, then* $r \mapsto^{\mathsf{PL}} r'$ *or* r *is a value* v*.*
4. *If* $\cdot \rhd t : A$*, then* $t \mapsto^{\mathsf{NL}} t'$ *or* t *is a term* $(v : A)$*.*

Finally, while we are discussing coverage, it is worth pointing out that the failure judgement also allows us to detect redundant or useless patterns, during the translation of ML patterns to focused patterns. As we compile each arm, we can check to see if the arm conjoined with the negation of its predecessor pattern fails, and if so, we know that the arm is redundant, and can signal an error.

6. Pattern Compilation

6.1 What Is Pattern Compilation?

If we implemented the focused pattern calculus in a naive way, we would observe that there are several inefficiencies in the pattern substitution algorithm, arising from and- and or-patterns. And-patterns $p \wedge p'$ can force the same value to be destructured multiple times. For example, a match of a value v against the pattern $[x \mid y] \wedge [u \mid v]$ will result in v being destructured twice — once to determine whether to take left or right branch for the pattern $[x \mid y]$, and again to determine whether to take the left or right branch for $[u \mid v]$. This is redundant, since one test suffices to establish whether v is a left- or right-injection.

Likewise, or-patterns can also introduce inefficiency, because their naive implementation is via backtracking — when a value v is matched against a pattern $p_1 \vee p_2$, we will try to match p_1 and if it fails, try matching p_2. This can result in repeated re-tests if the cause of the failure is deep within the structure of a term. For example, suppose we match a value of the form $\mathsf{inl}\ \mathsf{inl}\ v'$ against a pattern like $[[\bot \mid \top] \mid \top] \vee [[\top \mid \bot] \mid \bot]$. Here, a naive left-to-right backtracking algorithm will case analyze two levels deep before failing on the left branch, and then it will repeat that nested case analysis to succeed on the right branch.

To avoid these inefficiencies, we want a *pattern compilation algorithm*. That is, we want to take an otherwise complete and deterministic pattern and arm, and transform them into a form that does no backtracking and does not repeatedly analyze the same term, and whose behavior under pattern substitution is identical to the original pattern and arm.

We can state this constraint by restricting our language of patterns to one in which 1) failure and or-patterns do not occur, and 2) the only use of conjunctive patterns is in patterns of the form $x \wedge p$. The first conditions ensures that pattern substitution will never fail or backtrack, and the second condition ensures that and-patterns can never force the re-analysis of a term, since a variable pattern can only trigger a substitution. The target sublanguage of patterns is given in the following grammar:

$$\begin{array}{rcl} c & ::= & x \mid \langle\rangle \mid \langle c_1, c_2 \rangle \mid [] \mid [c_1 \mid c_2] \\ & \mid & \top \mid x \wedge [c_1 \mid c_2] \mid x \wedge \langle c_1, c_2 \rangle \end{array}$$

Observe that a restricted pattern c corresponds to a series of primitive let-bindings, pair bindings, and case statements on sums, each of which corresponds to a form in the usual lambda calculus.

So we can formalize the pattern compilation as asking: is there an algorithm that can take a complete, deterministic pattern p with arm r, and translate it into a pattern c (with arm r'), such that for all v, if ${}^{\llcorner}\langle\!\langle v/p \rangle\!\rangle_A\, r \hookrightarrow r''$ if and only if ${}^{\llcorner}\langle\!\langle v/p \rangle\!\rangle_A\, r' \hookrightarrow r''$?

6.2 The Pattern Compilation Algorithm

We give a pattern compilation algorithm in Figure 2, which takes two arguments as an input. The first is a pattern context Δ. This argument will be used as the termination metric for the function, and will get smaller at each recursive call. It also constrains the type of the second argument. The second argument is a set S of pairs, with each pair consisting of a row of patterns q_i and an arm r. Each row's pattern list is the same length as the metric argument, and the n-th pattern in the pattern list is either $\top$ or the same pattern as the n-th pattern in the metric list Δ. Additionally, this set satisfies the invariant that for any sequence of values $v_i : A_i$ of the right types, there is exactly one element of S for which all of the v_i can succeed in matching its q_i.

The set S is related to the matrix of patterns found in the traditional presentations of pattern compilation. We only need a set instead of an array, because our alternation operator $p \vee p'$ is a form of non-deterministic choice. However, this also means we need the extra invariant that there is only one matching row for any value, because we have no ordering that would let us pick the first matching row.

With this description in mind, we can state the correctness theorems below:

Proposition 12 (Soundness of Pattern Compilation). *If we have that*

- $\Delta = p_1 : A_1, \ldots, p_n : A_n$
- S *is a set of pairs such that for every* $(p'_1, \ldots, p'_n; r') \in S$, $p'_i \in \{p_i, \top\}$, *and* $\Gamma; p'_1 : A_1, \ldots, p'_n : A_n, \Delta' \rhd r' : C$
- *For all* $v_1 : A_1, \ldots, v_n : A_n$ *there exist* $(p'_1, \ldots, p'_n; r'_1) \in S$ *such that there exist* $r'_2 \ldots r'_{n+1}$ *such that for all* $i \in \{1 \ldots n\}$. ${}^{\mathsf{L}}\langle\!\langle v_i/p'_i \rangle\!\rangle_{A_i}\, r'_i \hookrightarrow r'_{i+1}$.
- $(c_1, \ldots, c_n; r_1) = \mathsf{Compile}(\Delta; S)$

then it holds that

- $\Gamma; c_1 : A_1, \ldots, c_n : A_n, \Delta' \rhd r'_1 : C$, *and*
- *For all* $v_1 : A_1, \ldots, v_n : A_n$, *if there exists a unique* $(p'_1, \ldots, p'_n; r_1) \in S$, *such that there exists a unique* $r_2, \ldots, r_{n+1}$, *such that for all* $1 \leq i \leq n$, ${}^{\mathsf{L}}\langle\!\langle v_i/p'_i \rangle\!\rangle_{A}\, r'_i \hookrightarrow r'_{i+1}$, *then there exist* $r_2 \ldots r_{n+1}$ *such that for all* $1 \leq i \leq n$, ${}^{\mathsf{L}}\langle\!\langle v_i/c_i \rangle\!\rangle_{A}\, r_i \hookrightarrow r_{i+1}$ *and* $r_{n+1} = r'_{n+1}$.

Proposition 13 (Termination of Pattern Compilation). *If we have that*

- $\Delta = p_1 : A_1, \ldots, p_n : A_n$
- *S is a set of pairs such that for every $(p'_1, \ldots, p'_n; r') \in S$, $p'_i \in \{p_i, \top\}$, and $\Gamma; p'_1 : A_1, \ldots, p'_n : A_n, \Delta' \rhd r' : C$*
- *For all $v_1 : A_1, \ldots, v_n : A_n$ there exists a unique $(p'_1, \ldots, p'_n; r'_1) \in S$ such that there exist unique $r'_2 \ldots r'_{n+1}$ such that for all $i \in \{1 \ldots n\}$. $^{\mathsf{L}}\langle\!\langle v_i/p'_i \rangle\!\rangle_{A_i}\, r'_i \hookrightarrow r'_{i+1}$*

then there is a $(c_1, \ldots, c_n; r_1) = \mathsf{Compile}(\Delta; S)$

Looking at Figure 2, we see that the compilation algorithm is recursive, and at each step it 1) breaks down the outermost constructor of the leftmost pattern in the pattern context Δ, 2) adjusts the set S to match the invariant, 3) recursively calls Compile, and 4) constructs the desired result from the return value.

Clearly, if Δ is empty, then there is no work to be done, and the algorithm terminates. If the first pattern is a unit pattern, then we know that the first element of each of the pattern lists in S is either $\top$ or $\langle\rangle$. So, we can adjust the elements of S by using inversion to justify dropping the $\top$ and $\langle\rangle$ patterns from the start of each of the pattern lists in S. Then, we can recursively call the pattern compiler, and use the unit introduction rule to justify restoring the unit pattern to the front of the list.

Likewise, if the first pattern is a pair pattern $\langle p_1, p_2 \rangle$, we can use inversion to justify splitting each of the pattern lists in S. If the first pattern in a pattern list is $\langle p_1, p_2 \rangle$, then we can send it to p_1, p_2, and if it is $\top$, we can send it to $\top, \top$.[4] (The intuition is that at a pair type, $\top$ is equivalent to $\langle \top, \top \rangle$.) Then, after the recursive call we can use pair introduction to construct the optimized pair pattern.

If the first pattern is an abort pattern $[]$, then we can return a pattern list that starts with $[]$ and is followed by a sequence of $\top$ patterns. This is fine because there are no values of type 0, so the correctness constraint holds vacuously.

We reach the first complex case when the first pattern is a sum pattern $[p_1 \mid p_2]$. First, we define the Left and Right functions, which take the elements of S and choose the left and right branches, respectively. (If the head of the pattern list is a $\top$ pattern, it gets assigned to both sides, since $(\top, qs; r)$ with the head at a sum type is equivalent to $([\top \mid \top], qs; [r \mid r])$.)

After splitting, we call $\mathsf{Compile}$ on the left and right sets. However, we can't just put them back together with a sum pattern, because we don't know that the tails of the two pattern lists are the same — they arise from different calls to $\mathsf{Compile}$. This is what the Merge function is for. Given two pattern contexts and arms consisting of optimized patterns, it will find a new pattern context and a new pair of arms, such that the new arms are substitution equivalent to the old ones, but which are typed under the new pattern list.

PROPOSITION 14 (Context Merging). *If we have that*

- $\Gamma; \Delta_1, c_1 : A, \Delta'_1 \rhd r_1 : C$,
- $\Gamma; \Delta_2, c_2 : A, \Delta'_2 \rhd r_2 : C$, *and*
- $\cdot \vdash v : A$,

then we may conclude that

- $(c'; r'_1; r'_2) = \mathsf{Merge}_A(\Delta_1; c_1; r_1; \Delta_2; c_2; r_2)$,
- $\Gamma; \Delta_1, c' : A, \Delta'_1 \rhd r'_1 : C$,
- $\Gamma; \Delta_2, c' : A, \Delta'_2 \rhd r'_2 : C$,
- ${}^{\mathsf{L}}\langle\!\langle v/c' \rangle\!\rangle^{\Delta_1}_{A}\ r'_1 \hookrightarrow r''_1$ *if and only if* ${}^{\mathsf{L}}\langle\!\langle v/c_1 \rangle\!\rangle^{\Delta_1}_{A}\ r_1 \hookrightarrow r''_1$, *and*
- ${}^{\mathsf{L}}\langle\!\langle v/c' \rangle\!\rangle^{\Delta_2}_{A}\ r'_2 \hookrightarrow r''_2$ *if and only if* ${}^{\mathsf{L}}\langle\!\langle v/c_2 \rangle\!\rangle^{\Delta_1}_{A}\ r_2 \hookrightarrow r''_2$

The notation Merge^* in Figure 2 indicates that we apply Merge to each element of the sequences cs_1 and cs_2. Its definition uses another auxilliary function Weaken. As its name might suggest, this is a generalized weakening lemma for the pattern context.

[4] Here, as in many of the other cases, we define a local function $\mathsf{split}((p; r))$ to perform this decomposition. Also, we write $\mathtt{map}\ \mathsf{split}(S)$ to indicate mapping over a set.

PROPOSITION 15 (Extended Weakening). *If* $\Gamma; \Delta, \Delta' \rhd r : C$ *and* $r' = \mathsf{Weaken}_A(\Delta; c; r)$*, then* $\Gamma; \Delta, c : A, \Delta' \rhd r' : C$*, and for all* $\cdot \vdash v : A$*,* $^{\mathsf{L}}\langle\!\langle v/c \rangle\!\rangle^{\Delta}_{A}\, r' \hookrightarrow r$*.*

If the first pattern is an and-pattern $p_1 \wedge p_2$, we can use inversion to justify splitting each of the pattern lists in S. If the first pattern in a pattern list is $p_1 \wedge p_2$, then we can send it to p_1, p_2, and if it is $\top$, we can send it to $\top, \top$. (As usual, the intuition is that $\top$ is equivalent to $\top \wedge \top$.) Then, after the recursive call we can use pair introduction to construct the optimized pair pattern. However, we're not quite done — our context has *two* patterns c_1 and c_2, and if we combined them with an and-pattern then we would potentially be forced to repeat tests, which we want to avoid. So we introduce an auxilliary function And, which satisfies the following property:

PROPOSITION 16 (Conjunction Simplification). *If we have that* $\Gamma; \Delta, c_1 : A, c_2 : A, \Delta' \rhd r : C$ *and* $\cdot \vdash v : A$*, then*

- $(c; r') = \mathsf{And}_A(\Delta; c_1; c_2; r)$*,*
- $\Gamma; \Delta, c : A, \Delta' \rhd r' : C$*, and*
- $^{\mathsf{L}}\langle\!\langle v/c \rangle\!\rangle^{\Delta}_{A}\, r' \hookrightarrow r''$ *if and only if* $^{\mathsf{L}}\langle\!\langle v/c_1 \wedge c_2 \rangle\!\rangle^{\Delta}_{A}\, r \hookrightarrow r''$*.*

It is worth looking at the pair pattern case in Figure 3 in a little more detail. It uses the exchange lemma to move the components of two pair patterns together – to change $\langle c_1, c_2 \rangle, \langle c_1', c_2' \rangle$ into c_1, c_1', c_2, c_2'', so that the subcomponents of the pair patterns can be conjoined together. This illustrates why it was necessary to prove that substitution and the deep pattern operations could commute: changing this order implies changing the order that the patterns substitutions would be performed.

This function follows the structure of the two pattern arguments, and when it reaches the pair-pair or sum-sum cases, it uses the deep inversion, introduction, and exchange algorithms to reorder the pattern hypotheses so that the tests can be merged.

Finally, we reach the case where the head of the pattern context is the or-pattern $p_1 \vee p_2$. For each element of S, the splitting algorithm proceeds as follows. Each element of the form $(p_1 \vee p_2, \ldots; r_1 \vee r_2)$ becomes two cases — $(p_1, \top, \ldots; r_1)$ and $(\top, p_2, \ldots; r_2)$. We know via inversion that $(p_1, \ldots; r_1)$ and $(p_2, \ldots; r_2)$ are well-typed, and using top-introduction makes each element satisfy the compile invariant for $p_1 : A, p_2 : A, \ldots$. Furthermore, any sequence of values $v, v, \ldots$ with a repeated value v will match one of these two elements exactly when the original would match $v, \ldots$.

So we can invoke $\mathsf{Compile}$ recursively, and receive $(c_1, c_2, \ldots; r)$ as a return value. As in the and-pattern case, we can combine the c_1 and c_2 patterns with the And function to find a c that does the job.

With the correctness of the compilation algorithm established, it is easy to show that the coverage algorithm of the previous section permits us to confidently invoke the pattern compiler.

PROPOSITION 17 (Pattern Compilation). *If we have that*

- p covers A *and* p det A,
- $\Gamma; p : A, \Delta \triangleright r : C$, *and*
- $\cdot \vdash v : A$,

then it is the case that

- $(c; r') = \mathsf{Compile}(p : A; \{(p, r)\})$,

- $^{\mathsf{L}}\langle\!\langle v/p \rangle\!\rangle_A \, r \hookrightarrow r''$ *if and only if* $^{\mathsf{L}}\langle\!\langle v/c \rangle\!\rangle_A \, r' \hookrightarrow r''$

7. Extensions and Future Work

This language is highly stylized, but some obvious extensions work out very easily. First, adding iso-recursive types $\mu\alpha.A$ is straightforward. If we have a term $\mathsf{roll}(e)$ as the constructor for a recursive

```
Compile$(\cdot; \{(\cdot; r)\}) \equiv$
    $(\cdot; r)$
Compile$(\langle\rangle : 1, \Delta; S) \equiv$
    let split$((\top, qs; r)) = \{(qs; r)\}$
        split$((\langle\rangle, qs; r)) = \{(qs; r)\}$
    let $(cs; r) =$ Compile$(\Delta; \bigcup \text{map split}(S))$
    $(\langle\rangle, cs; r)$
Compile$(\langle p_1, p_2\rangle : A_1 \times A_2, \Delta; S) \equiv$
    let split$((\top, qs; r)) = \{(\top, \top, qs; r)\}$
        split$((\langle p_1, p_2\rangle, qs; r)) = \{(p_1, p_2, qs; r)\}$
    let $(c_1, c_2, cs; r) =$ Compile$(p_1 : A_1, p_2 : A_2, \Delta; \bigcup \text{map split}(S))$
    $(\langle c_1, c_2\rangle, cs; r)$
Compile$([] : 0, \overrightarrow{p : A}; S) \equiv$
    $([] : 0, \overrightarrow{\top}; [])$
Compile$([p_1 \mid p_2] : A_1 + A_2, \overrightarrow{p : A}; S) \equiv$
    let Left$((\top, \overrightarrow{p'}; r)) = \{(\top, \overrightarrow{p'}; r)\}$
        Left$(([p_1 \mid p_2], \overrightarrow{p'}; [r_1 \mid r_2])) = \{(p_1, \overrightarrow{p'}; r_1)\}$
    let Right$((\top, \overrightarrow{p'}; r)) = \{(\top, \overrightarrow{p'}; r)\}$
        Right$(([p_1 \mid p_2], \overrightarrow{p'}; [r_1 \mid r_2])) = \{(p_2, \overrightarrow{p'}; r_2)\}$
    let $(c_1, \overrightarrow{cs_1}; r_1) =$ Compile$(p_1 : A_1, \overrightarrow{p : A}; \bigcup \text{map Left}(S))$
    let $(c_2, \overrightarrow{cs_2}; r_2) = \mathsf{Compile}(p_2 : A_2, \overrightarrow{p : A}; \bigcup \mathsf{map\ Right}(S))$
    let $(cs; r'_1; r'_2) = \mathsf{Merge}^*_{\vec{A}}(c_1 : A_1; \overrightarrow{cs_1}; r_1; c_2 : A_2; \overrightarrow{cs_2}; r_2)$
    $([c_1 \mid c_2], \overrightarrow{cs}; [r'_1 \mid r'_2])$
$\mathsf{Compile}(x : A, \Delta; S) \equiv$
    let $\mathsf{split}((\top, qs; r)) = \{(qs; r)\}$
        $\mathsf{split}((x, qs; r)) = \{(qs; r)\}$
    let $(cs; r) = \mathsf{Compile}(\Delta; \bigcup \mathsf{map\ split}(S))$
    $(x, cs; r)$
$\mathsf{Compile}(\top : A_1, \Delta; S) \equiv$
    let $\mathsf{split}((\top, qs; r)) = \{(qs; r)\}$
    let $(cs; r) = \mathsf{Compile}(\Delta; \bigcup \mathsf{map\ split}(S))$
    $(\top, cs; r)$
$\mathsf{Compile}(p_1 \wedge p_2 : A_1, \Delta; S) \equiv$
    let $\mathsf{split}((\top, qs; r)) = \{(\top, \top, qs; r)\}$
        $\mathsf{split}((p_1 \wedge p_2, qs; r)) = \{(p_1, p_2, qs; r)\}$
    let $(c_1, c_2, cs; r) = \mathsf{Compile}(p_1 : A_1, p_2 : A_1, \Delta; \bigcup \mathsf{map\ split}(S))$
    let $(c; r') = \mathsf{And}_A(\cdot; c_1; c_2; r)$
    $(c, cs; r')$
$\mathsf{Compile}(\bot : A_1, \Delta; S) \equiv$
    let $\mathsf{split}((\top, qs; r)) = \{(qs; r)\}$
        $\mathsf{split}((\bot, qs; r)) = \emptyset$
    let $(cs; r) = \mathsf{Compile}(\Delta; \bigcup \mathsf{map\ split}(S))$
    $(\top, cs; r)$
$\mathsf{Compile}(p_1 \vee p_2 : A_1, \Delta; S) \equiv$
    let $\mathsf{split}((\top, qs; r)) = \{(\top, \top, qs; r)\}$
        $\mathsf{split}((p_1 \vee p_2, qs; r_1 \vee r_2)) = \{(p_1, \top, qs; r_1), (\top, p_2, qs; r_2)\}$
    let $(c_1, c_2, cs; r) = \mathsf{Compile}(p_1 : A_1, p_2 : A_1, \Delta; \bigcup \mathsf{map\ split}(S))$
    let $(c; r') = \mathsf{And}_A(\cdot; c_1; c_2; r)$
    $(c, cs; r')$
```

Figure 2. Pattern Compilation

type, then we can simply add a pattern $\mathsf{roll}(p)$ to the pattern languge. We believe supporting System-F style polymorphism is also straightforward. Universal quantification $\forall\alpha.\ A$ is a negative connective, and so does not interact with the pattern language. Existential quantification $\exists\alpha.\ A$ is a positive connective with introduction form $\mathsf{pack}(A, e)$, and we can add a pattern $\mathsf{pack}(\alpha, p)$ to support its elimination.

Features such as GADTs (Jones et al. 2006; Simonet and Pottier 2007) and pattern matching for dependent types (Coquand 1992; Xi 2003; McBride 2003), are much more complicated. In both of these cases, matching against a term can lead to the discovery of information that refines the types in the rest of the match. This is a rather subtle interaction, and deserves further study.

```
$\mathsf{And}_A(\Delta; \top; c; r) = \mathsf{And}_A(\Delta; c; \top; r) =$
    $(c; r)$
$\mathsf{And}_A(\Delta; x \wedge c_1; c_2; r) = \mathsf{And}_A(\Delta; c_1; x \wedge c_2; r) =$
    $\mathsf{let}\ (c'; r') = \mathsf{And}_A(\Delta; c_1; c_2; r)$
    $\mathsf{if}\ c' = y \wedge c''\ \mathsf{then}\ (c'; [x/y]r')\ \mathsf{else}(x \wedge c'; r')$
$\mathsf{And}_1(\Delta; \langle\rangle; x; r) = \mathsf{And}_1(\Delta; x; \langle\rangle; r) =$
    $(x; r)$
$\mathsf{And}_0(\Delta; []; x; r) = \mathsf{And}_0(\Delta; x; []; r) =$
    $([]; \mathsf{Abort}_0(\Delta))$
$\mathsf{And}_{A \times B}(\Delta; x; \langle c_1, c_2\rangle; r) = \mathsf{And}_{A \times B}(\Delta; \langle c_1, c_2\rangle; x; r) =$
    $(x \wedge \langle c_1, c_2\rangle; r)$
$\mathsf{And}_{A+B}(\Delta; x; [c_1 \mid c_2]; r) = \mathsf{And}_{A+B}(\Delta; [c_1 \mid c_2]; x; r) =$
    $(x \wedge [c_1 \mid c_2]; r)$
$\mathsf{And}_A(\Delta; x; y; r) =$
    $(x; [x/y]r)$
```

```
$\mathsf{And}_1(\Delta; \langle\rangle; \langle\rangle; r) =$
    $(\langle\rangle; r)$
$\mathsf{And}_0(\Delta; []; []; r) =$
    $([]; \mathsf{Abort}_0(\Delta))$
$\mathsf{And}_{A \times B}(\Delta; \langle c_1, c_2\rangle; \langle c'_1, c'_2\rangle; r) =$
    $\mathsf{let}\ (c''_1; r') = \mathsf{And}_A(\Delta; c_1; c'_1; \mathsf{Ex}(\Delta, c_1 : A; c_2 : B; c'_1 : A; r))$
    $\mathsf{let}\ (c''_2; r'') = \mathsf{And}_B(\Delta; c_2; c'_2; r')$
    $(\langle c''_1, c''_2\rangle; r'')$
$\mathsf{And}_{A+B}(\Delta; [c_1 \mid c_2]; [c'_1 \mid c'_2]; r) =$
    $\mathsf{let}\ (c''_1; r''_1) = \mathsf{And}_A(\Delta; c_1; c'_1; \mathsf{Out}^{\mathsf{L}}_{+}(\Delta, c_1 : A; \mathsf{Out}^{\mathsf{L}}_{+}(\Delta; r)))$
    $\mathsf{let}\ (c''_2; r''_2) = \mathsf{And}_B(\Delta; c_2; c'_2; \mathsf{Out}^{\mathsf{R}}_{+}(\Delta, c_2 : A; \mathsf{Out}^{\mathsf{R}}_{+}(\Delta; r)))$
    $([c''_1 \mid c''_2]; \mathsf{Join}_{+}(\Delta; r''_1; r''_2))$
```

Figure 3. Conjunction Simplification

Another direction is to treat the proof term assignment discussed here as a lambda calculus in and of itself, rather than as a programming language, as we have done here. For example, we can use the exchange function to generalize the pattern substitution to deal with open terms and study properties like confluence and normalization. The presence of sums makes this a trickier question than it may seem at first glance; focusing seems to eliminate the need for some, but not all, of the commuting conversions for sum types.

Finally, we should more carefully study the implications for implementation. A simple ML implementation of the algorithms given in this paper can be found at the author's web site, but it is difficult to draw serious conclusions from it because so little engineering effort has gone into it.

However, a few prelimary observations are possible. Implementing coverage checking is very easy – it is roughly 80 lines of ML code. This closely follows the inference rules given above, with the the addition of memoization to avoid repeatedly trying to find failure derivations of the same sequence of patterns. This seems to suffice for the sorts of patterns we write by hand; it is not clear whether there are (easily-avoidable) pathological cases that machine-generated programs might exhibit. A more-or-less direct transliteration of the pattern compiler is in the neighborhood of 300 lines of ML. While it is reasonably fast on small hand-written examples, it should probably not be used in a production compiler. In particular, the Merge$*$ algorithm is implemented via iterated calls to Merge, which can result in an enormous number of redundant traversals of the pattern tree. This was done to simplify the correctness proof, but a production implementation should avoid that.

8. Related Work

We first learned to view pattern matching as arising from the invertible left rules of the sequent calculus due to the work of Kesner et al. (1996), and Cerrito and Kesner (2004). We have extended their work by building on a focused sequent calculus. This permits

```
$\mathsf{Merge}_A(\Delta_1; \top; r_1; \Delta_2; c_2; r_2) =$
$\mathsf{Merge}_A(\Delta_2; c_2; r_2; \Delta_1; \top; r_1) =$
    let $r_1' = \mathsf{Weaken}_A(\Delta_1; c_2; r_1)$
    $(c_2; r_1'; r_2)$
$\mathsf{Merge}_A(\Delta_1; x; r_1; \Delta_2; c_2; r_2) =$
$\mathsf{Merge}_A(\Delta_2; c_2; r_2; \Delta_1; x; r_1) =$
```

```
    let $r'_1 = \mathsf{Weaken}_A(\Delta_1; c_2; r_1)$
    let $(c; r'_2) = \mathsf{And}_A(\Delta_2; x; c_2; r_2)$
    $(c; r'_1; r'_2)$
$\mathsf{Merge}_A(\Delta_1; x \wedge c_1; r_1; \Delta_2; c_2; r_2) =$
$\mathsf{Merge}_A(\Delta_2; c_2; r_2; \Delta_1; x \wedge c_1; r_1) =$
    let $(c'; r'_1; r'_2) = \mathsf{Merge}_A(\Delta_1; c_1; r_1; \Delta_2; c_2; r_2)$
    let $(c''; r''_1) = \mathsf{And}_A(\Delta_1; x; c'_1; r'_1)$
    let $(\_; r''_2) = \mathsf{And}_A(\Delta_2; x; c'_2; r'_2)$
    $(c''; r''_1; r''_2)$
$\mathsf{Merge}_1(\Delta_1; \langle\rangle; r_1; \Delta_2; \langle\rangle; r_2) =$
    $(\langle\rangle; r_1; r_2)$
$\mathsf{Merge}_0(\Delta_1; []; r_1; \Delta_2; []; r_2) =$
    $([]; \mathsf{Abort}_0(\Delta_1); \mathsf{Abort}_0(\Delta_2))$
$\mathsf{Merge}_{A\times B}(\Delta_1; \langle c_1, c_2\rangle; r_1; \Delta_2; \langle c'_1, c'_2\rangle; r_2) =$
    let $(c''_1; r'_1; r'_2) = \mathsf{Merge}_A(\Delta_1; c_1; r_1; \Delta_2; c'_1; r_2)$
    let $(c''_2; r''_1; r''_2) = \mathsf{Merge}_B(\Delta_1, c''_1 : A; c_2; r'_1; \Delta_2, c''_1 : A; c'_2; r'_2)$
    $(\langle c''_1, c''_2\rangle; r''_1; r''_2)$
$\mathsf{Merge}_{A+B}(\Delta_1; [c_1 \mid c_2]; r_1; \Delta_2; [c'_1 \mid c'_2]; r_2) =$
    let $(c''_1; r'_1; r'_2) = \mathsf{Merge}_A(\Delta_1; c_1; \mathsf{Out}^{\mathsf{L}}_{+}(\Delta_1; r_1); \Delta_2; c'_1; \mathsf{Out}^{\mathsf{L}}_{+}(\Delta_2; r_2))$
    let $(c''_2; r''_1; r''_2) = \mathsf{Merge}_B(\Delta_1; c_2; \mathsf{Out}^{\mathsf{R}}_{+}(\Delta_1; r_1); \Delta_2; c'_2; \mathsf{Out}^{\mathsf{R}}_{+}(\Delta_2; r_2))$
    $([c''_1 \mid c''_2]; \mathsf{Join}_{+}(\Delta_1; r'_1; r''_1); \mathsf{Join}_{+}(\Delta_2; r'_2; r''_2))$

$\mathsf{Weaken}_A(\Delta; \top; r) = r$
$\mathsf{Weaken}_A(\Delta; x; r) = r$
$\mathsf{Weaken}_1(\Delta; \langle\rangle; r) = r$
$\mathsf{Weaken}_0(\Delta; []; r) = \mathsf{Abort}_0(\Delta)$
$\mathsf{Weaken}_A(\Delta; x \wedge c; r) = \mathsf{Weaken}_A(\Delta; c; r)$
$\mathsf{Weaken}_{A\times B}(\Delta; \langle c_1, c_2\rangle; r) =$
  $\mathsf{Weaken}_B(\Delta, c_1 : A; c_2; \mathsf{Weaken}_A(\Delta; c_1; r))$
$\mathsf{Weaken}_{A+B}(\Delta; [c_1 \mid c_2]; r) =$
  $\mathsf{Join}_{+}(\Delta; \mathsf{Weaken}_A(\Delta; c_1; r); \mathsf{Weaken}_B(\Delta; c_2; r))$
```

Figure 4. Pattern Merge

us to give a simpler treatment; the use of an ordered context allows us to eliminate the communication variables they used to link sum patterns and their bodies. Furthermore, our failure and nondeterministic choice patterns permit us to explain the sequential pattern matching found in functional languages, coverage checking, and pattern compilation.

Focusing was introduced by Andreoli (1992), in order to constrain proof search for linear logic. Pfenning (in unpublished lecture notes) gives a simple focused calculus for intuitionistic logic, and Liang and Miller (2007) give calculi for focused intuitionistic logic, which they relate to both linear and classical logic. Neither of these have proof terms. Zeilberger (2007) gives a focused calculus based on Dummett's notion of logical harmony (Dummett 1991). This calculus does not have a coverage algorithm; instead coverage is a side-condition of his typing rules.

Our pattern substitution is a restricted form of hereditary substitution, which Watkins et al. (2004) introduced as a way of reflecting the computational content of structural proofs of cut admissibility (Pfenning 2000).

In his work on Ludics, Girard (2001) introduced the idea of the *daimon*, a sequent which corresponds to a failed proof. Introducing such sequents can give a logical calculus certain algebraic closure properties, at the cost of soundness. However, once the requisite properties have been used, we can verify that we have any given proof is genuine by checking that the undesirable sequents are not present. This is an idea we exploited with the introduction of the $\perp$ and $p_1 \vee p_2$ patterns, which make our language of patterns closed under complement.

Zeilberger (2008) gives a higher-order focused calculus. In this style of presentation, the inversion judgement is given as a single infinitary rule, defined using the functions of the ambient metalogic, rather than the explicit collection of rules we gave. The virtue of their approach is that it defers questions of coverage and decomposition order into the metalogic. However, this is precisely the question we wanted to explicitly reason about.

In real compilers, there are two classical approaches to compiling pattern matching, either by constructing a decision tree (described by Cardelli (1984) and Pettersson (1992)) or building a backtracking automaton (described by Augustsson (1985)). Our calculus uniformly represents both approaches, since backtracking can be represented with the use of the nondeterministic disjunction pattern $p_1 \vee p_2$ and the abort pattern $[]$, and case splitting is represented with the sum-pattern $[p_1 \mid p_2]$. This lets us view pattern compilation as a source-to-source transformation, which simplifies the correctness arguments.

Fessant and Maranget (2001) describe a modern algorithm for pattern compilation which operates over matrices of patterns. Their algorithm tries to make use of an efficient mix of backtracking and branching, whereas our compilation algorithm builds a pure decision tree. It might be possible to find a presentation of their ideas without having to explicitly talk about low-level jumps and gotos, by replacing $p \vee p'$ with a biased choice that always tries p first.

Maranget (2007) also describes an algorithm for generating warnings for non-exhaustive matches and useless clauses. This algorithm is a specialized version of the decision tree compilation algorithm which returns a boolean instead of a tree. However, his correctness proof is not as strong as ours: Maranget defines a matching relation and shows that a complete pattern will always succeed on a match, but the connection between the matching relation and the language semantics is left informal.

Sestoft (1996) shows how to generate pattern matching code via partial evaluation. This ensures the correctness of the compilation, but he does not consider the question of coverage checking.

Jay (2004) has also introduced a pattern calculus. Roughly, he takes the view that datatypes are just subsets of the universe of program terms (like Prolog's Herbrand universe), and then allows defining programs to match on the underlying tree representations of arbitrary data. This approach to pattern matching is very expressive, but its extremely intensional nature means its compatibility with data abstraction is unclear.

The work on the ρ-calculus (Cirstea and Kirchner 2001) is another general calculus of pattern matching. It treats terms similarly to Jay's pattern calculus. Additionally, it uses the success or failure of matching as a control primitive, similar to the way that false- and or-patterns work in this work. However, the focus in this paper was on the case where the nondeterminism is inessential, rather than exploring its use as a basic control mechanism.

Acknowledgements. The author thanks Jonathan Aldrich, Robert Harper, Dan Licata, William Lovas, Frank Pfenning, Jason Reed, John Reynolds, Kevin Watkins, and Noam Zeilberger for encouragement and advice. This work was supported in part by NSF grant CCF-0541021, NSF grant CCF-0546550, DARPA contract HR00110710019 and the Department of Defense.

References

J.M. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation*, 2(3):297, 1992.

L. Augustsson. Compiling pattern matching. *Proc. of a conference on Functional Programming Languages and Computer Architecture*, pages

368–381, 1985.

Luca Cardelli. Compiling a functional language. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, pages 208–217, New York, NY, USA, 1984. ACM Press. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802037.

S. Cerrito and D. Kesner. Pattern matching as cut elimination. *Theoretical Computer Science*, 323(1-3):71–127, 2004.

H. Cirstea and C. Kirchner. The rewriting calculus - Part I. *Logic Journal of the IGPL*, 9(3): 2001.

T. Coquand. Pattern matching with dependent types. *Proceedings of the Workshop on Types for Proofs and Programs*, pages 71–83, 1992.

M. Dummett. *The Logical Basis of Metaphysics*. Duckworth, 1991.

Fabrice Le Fessant and Luc Maranget. Optimizing pattern matching. In *ICFP '01: Proceedings of the sixth ACM SIGPLAN International Conference on Functional Programming*, pages 26–37, New York, NY, USA, 2001. ACM Press. ISBN 1-58113-415-0. doi: http://doi.acm.org/10.1145/507635.507641.

J.Y. Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(03):301–506, 2001.

C. B. Jay. The pattern calculus. *Transactions on Programming Languages and Systems* 26(6):911-937, 2004.

S.P. Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. *Proceedings of the eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, 2006.

D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. *Information and Computation*, 124(1):32–61, 1996.

Chuck Liang and Dale Miller. Focusing and polarization in intuitionistic logic. In *16th EACSL Annual Conference on Computer Science and Logic*. Springer-Verlag, 2007. URL http://www.cs.hofstra.edu/~csccl/focusil.pdf.

Luc Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 2007.

C. McBride. Epigram. *Types for Proofs and Programs*, 3085:115–129, 2003.

Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Ann. Pure Appl. Logic*, 51(1-2):125–157, 1991.

R.R. Milner, Mads Tofte, Robert Harper, and David McQueen. *The Definition of Standard ML:(revised)*. MIT Press, 1997.

Mikael Pettersson. A term pattern-match compiler inspired by finite automata theory. In Uwe Kastens and Peter Pfahler, editors, *CC*, volume 641 of *Lecture Notes in Computer Science*, pages 258–270. Springer, 1992. ISBN 3-540-55984-1.

F. Pfenning. Structural Cut Elimination I. Intuitionistic and Classical Logic. *Information and Computation*, 157(1-2):84–141, 2000.

P. Sestoft. ML pattern match compilation and partial evaluation. *Lecture Notes in Computer Science*, 1110:446, 1996. URL citeseer.ist.psu.edu/sestoft96ml.html.

Vincent Simonet and Francois Pottier. A constraint-based approach to guarded algebraic data types. *Transactions on Programming Languages and Systems* 29(1), 2007.

K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework: The propositional fragment. *Types for Proofs and Programs*, pages 355–377, 2004.

H. Xi. Dependently Typed Pattern Matching. *Journal of Universal Computer Science*, 9(8):851–872, 2003.

Noam Zeilberger. The logical basis of evaluation order. Thesis proposal, May 2007. Carnegie Mellon, Pittsburgh, Pennsylvania. Available at http://www.cs.cmu.edu/~noam/research/proposal.pdf., 2007.

Noam Zeilberger. Focusing and higher-order abstract syntax. In George C. Necula and Philip Wadler, editors, *POPL*, pages 359–369. ACM, 2008. ISBN 978-1-59593-689-9.