Focusing on Pattern Matching

Neelakantan R. Krishnaswami

Carnegie Mellon University neelk@cs.cmu.edu

Abstract

We show how to extend the Curry-Howard correspondence to pattern matching, by showing how it arises as a natural proof term assignment to a focused sequent calculus for propositional logic. We demonstrate the value of this calculus by deriving a simple, novel algorithm to check the exhaustiveness of pattern matching, and to reconstruct a program transformation corresponding to compiling patterns via decision trees.

Categories and Subject Descriptors F.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Pattern Matching, Curry-Howard, Coverage checking, Pattern compilation, Focusing, Sequent calculus

1. Introduction

The Curry-Howard correspondence between typed lambda calculi and systems of intuitionistic logic is one of the most elegant ideas in the theory of functional programming languages. Pattern matching is one of the most practically useful features of modern functional programming languages. However, these two ideas sit uneasily together; when programs are considered as proofs, we usually make the informal claim that pattern matching corresponds to a case analysis in a proof, but this is not often given a precise mathematical formalization.

The absence of such a formalization relegates important features such as coverage checking and match redundancy tests to the status of implementation detail, buried within the depths of a compiler. For example, the Definition of Standard ML [13] specifies that coverage checking will be performed in English text, and does not give a precise characterization of what that means.

Consider the following pseudo-ML program:

Са	ase e	of				
Ι	(Inl	x,	Inl	y)	->	e1
Ι	(Inr	u,	Inr	v)	->	e2
Τ	_				->	e3

Even though programmers write things like this every day, it is an expression with surprising depths. This program merges multiple logical eliminations and makes use of the sequential priority ordering of ML pattern matching, to ensure that the first two clauses take priority over the final wildcard. Programmers also expect that

[Copyright notice will appear here once 'preprint' option is removed.]

we will warn them if there are no coverage errors, and conversely also warn them if any of the clauses turn out to be useless. Our contributions are the following:

- First, we give a strongly logically-motivated nondeterministic calculus of pattern matching, arising from the proof-theoretic notion of focusing. We then show how ML-like patterns can be encoded in this language, including features like or-patterns and the left-to-right sequentiality of ML-style pattern matching. This calculus also extends easily to features like recursive and existential types.
- Second, we give a simple inductive characterization of when a pattern is deterministic and exhaustive, and prove this algorithm is sound. This gives a novel exhaustiveness test that does not rely on examining the output of compiling a set of patterns to decision trees.
- Third, we show how to take programs with nondeterministic
 patterns transform them into an equivalent version that makes
 no use of nondeterminism, and prove it correct. This corresponds to compiling pattern matching via decision trees, and
 shows how the two main approaches of compiling pattern
 matching (backtracking and decision trees) can be related.

2. Core Type Theory

Our programming language can be viewed as a bidirectionallytyped proof term assignment for a *focused* sequent calculus for intuitionistic logic. Focusing was originally introduced by Andreoli [1] to reduce the number of trivially-different in the sequent calculus, and thereby give normal forms for cut free proofs in linear logic.

The key idea underlying focusing is to divide the logical connectives into two groups, based on an analysis of whether the connective's left or right rules are invertible. One group, the positive connectives, has invertible left rules, and the other group, the negative connectives, has invertible right rules.¹ Proof search can exploit this categorization, since invertible rules can be applied in any order without the need for backtracking, and once the inversion phase is complete, we can apply a chain of non-invertible rules.

Now, we can express any pattern match as a nested sequence of eliminations of sums and products, but there are usually multiple nestings, and programmers are indifferent to which one is chosen. Concretely, consider our example from the introduction:

case e of
| (Inl x, Inl y) -> e1
| (Inr u, Inr v) -> e2
| _ -> e3

¹ Andreoli called the positive and negative connectives are called *synchronous* and *asynchronous*, respectively.

This expression could be written using primitive case statements in two ways, with no reason to choose between either:

```
let (a, b) = e in
case(a, Inl x. case(b, Inl y. e1, Inr _. e3),
     Inr u. case(b, Inl _. e3, Inr v. e2))
let (a, b) = e in
```

The main observation underlying our type theory is that these trivial reorderings correspond exactly to the kinds of uninteresting distinctions that focusing eliminates.

In intuitionistic logic, the implication connective is negative, and the sum type is positive. The product can be understood either way, since both its left and right rules are invertible. Understood as a negative connective, it has projective eliminations, and as a positive connective, it has a binding elimination:

$$\frac{\Gamma \vdash e: A_1 \times A_2}{\Gamma \vdash \pi_i \ e: A_i} \times \text{-Negative}$$

$$\frac{\Gamma \vdash e: A_1 \times A_2}{\Gamma \vdash \text{let} \ (x, y) = e \text{ in } e': B} \times \text{-Positive}$$

Since we want to interleave sum and product eliminations, we will choose to treat products as a positive connective. So the function space $(A \rightarrow B)$ is the only negative connective; and sums (A + B), products $(A \times B)$ and their units (0 and 1) are positive connectives

We give the syntax of our language in Figure 1, and the typing rules in Figure 2. The types are the usual types of the simply-typed lambda calculus, and we divide the syntax of program expressions into four classes, for the introduction and elimination forms for positive and negative types, respectively.

The introduction forms for positive types are the usual unit $\langle \rangle$, pair $\langle e_1, e_2 \rangle$, and left and right injections, inl e and inr e. They are typed with the judgement $\Gamma \vdash e : A$, which reads that "given a variable context Γ , *e* typechecks at type *A*".

The introduction form for the negative type is a lambda abstraction λp . u. The negative introductions are typechecked using the judgement $\Gamma; \Delta \vdash u : A$, which reads "given a variable context Γ and the ordered pattern context Δ , u typechecks at type A". Note that instead of binding a variable, a lambda abstraction binds an entire pattern, which goes into the right end of the pattern context. The existence of the pattern context means that patterns are separated from the bodies of their arms, unlike a case expression in an ML program.

As a result, the judgement for typing positive eliminations, $\Gamma; \Delta \triangleright r : A$, must link the branches in pattern context with the arms of the expression r. This is why the assumption that the pattern context is ordered is essential - it allows us to link branching in the patterns with branching in the arms. This judgement's job is to decompose the patterns and move their variables to Γ , and produces variables out of them, and it does so by systematically breaking on the leftmost pattern in the context.

The variable pattern x binds values of any type, and its typing rule simply moves the variable from Δ to Γ .

We have the variable $\langle \rangle$ pattern for unit values, whose typing rule 1L, says that an expression is typeable with a unit pattern, if it is typeable without it. This makes sense, given the intuition that matching the unit pattern binds no variables. Likewise, the rule for the pattern $\langle p_1, p_2 \rangle$ matching values of product type, can be seen as justified when matching destructures a pair given to it and matches the left and right halves against p_1 and p_2 respectively.

The pattern for sum types is [inl p_1 | inr p_2]. Here, we depart from the ML/Haskell style by requiring that a programmer to supply patterns describing both the left and the right injections. This ensures that the sum pattern is inherently complete – when a value of the form inl v is matched against [inl p_1 | inr p_2], we take the left branch and then match v against p_1 , and symmetrically for inr v. Since this is a branch, the proof term for this rule requires that we have two arms $[r_1 \mid r_2]$ to account for the left and right possibilities. Finally, we have the elimination pattern [] for the empty type 0. Since there are no branches in the pattern, there are no branches in its proof term, which is also [].

We also have a pattern \top , which corresponds to the wildcard pattern _ in ML, which discards its value argument. As a result, the premise of the typing rule $\top L$ asks that the expression be welltyped without the \top hypothesis. The conjunctive pattern $p_1 \wedge p_2$ is a generalization of the as-patterns of ML. Its semantics are to match a value against p_1 , and then to match that same value against p_2 .

Now, we come to the two strangest members of our pattern language. The disjunctive choice pattern $p_1 \vee p_2$ matches a value against either p_1 or p_2 , nondeterministically. This is an "angelic" choice, in the sense that this match can fail if and only if the value fails to match both p_1 and p_2 . So we therefore need a branching proof term $r_1 \vee r_2$, corresponding to the two alternatives. The failure pattern \perp is a pattern that fails to match any value, and has a proof term \perp . This syntactically internalizes match failure in the language of patterns.

Once all of the hypotheses in the pattern context are eliminated, we can either shift to one of the non-invertible phases (the positive introductions or the negative eliminations), or case analyze an expression with $case(t, p \Rightarrow r)$. Here, t supplies the expression to analyze, with p as the pattern and r as the arms.

Notice that the linearity constraint on the variables appearing in a pattern arises naturally from requiring that we adopt the Barendregt on variable names - if we are obliged to choose names so that there are never any repetitions of a variable in Γ , it is not possible to write repeated variables in any particular branch of a pattern. That is, we cannot write $\langle x, x \rangle$, because the second would put two x's into the context Γ . However, we can write [inl x | inr x], because each x will be directed into a different branch of the proof tree.

Finally, we can look at the negative eliminations t, typed with the typing judgement $\Gamma \triangleright t : A$. In addition to variable references x and function applications t e, we also include explicitly typeannotated expressions (e: A) in this category.

As an aside, our type system is not precisely a focused calculus. If it were, it would only type beta-normal, eta-long terms. To do this, we would have to change our rules to eliminate the term (e: A) (which corresponds to a use of the Cut rule), require all variables would have to be of negative type (to enforce eta-long forms at positive types), and require the negative elimination rule would have to go all the way down to positive types (to enforce eta-long forms at negative types). However, since we are interested in programming with this language, we relax these conditions in order to allow programs that can actually reduce into our system.

Each context gives rise to an associated substitution principle, and substitution for variables is given in Figure 3. The only deviation from conventional practice is that we carry along a type, and add an annotation when we substitute an expression for a variable.

LEMMA 1 (Substitution). If $\Gamma \vdash e : A$ then

- if $\Gamma, x : A \vdash e' : B$ then $\Gamma \vdash [e/x]_A^{\mathsf{Pl}} e' : B$
- if $\Gamma, x: A; \Delta \vdash u: B$ then $\Gamma; \Delta \vdash [e/x]_A^{\mathsf{NI}} u: B$ if $\Gamma, x: A; \Delta \vdash r: B$ then $\Gamma; \Delta \vdash [e/x]_A^{\mathsf{NI}} u: B$ if $\Gamma, x: A; \Delta \triangleright r: B$ then $\Gamma; \Delta \triangleright [e/x]_A^{\mathsf{PE}} r: B$ if $\Gamma, x: A \triangleright t: B$ then $\Gamma \triangleright [e/x]_A^{\mathsf{NE}} t: B$

The informal explanation of pattern matching given above is formalized in the pattern substitution judgements $\langle\!\langle v/p \rangle\!\rangle_A^{PE} r \sim r'$ and $\langle\!\langle v/p \rangle\!\rangle_A^{NI} u \sim u'$, which are given in Figure 4. The pattern substitution we give here is a restricted form of the notion of hereditary substitution [16], and corresponds to the computational content of a proof of cut-admissibility.

Thanks to the disjunctive pattern $p_1 \vee p_2$ and the failing pattern \perp , pattern substitution forms a relation rather than a function. Sometimes a value will fail to match against a well-typed pattern p, and against other patterns it might match multiple ways. Nevertheless there is a substitution principle for pattern substitution:

LEMMA 2 (Pattern Substitution). If $\cdot \vdash v : A$, then

- if $\Gamma; p: A, \Delta \vdash u : B$ and $\langle\!\langle v/p \rangle\!\rangle_A^{NI} u \rightsquigarrow u'$, then we have $\Gamma: \Delta \vdash u' : B.$
- if $\Gamma; p: A, \Delta \rhd r : B$ and $\langle\!\langle v/p \rangle\!\rangle_A^{PE} r \rightsquigarrow r'$, then we have $\Gamma; \Delta \rhd r' : B$.

We give an operational semantics for this language in Figure 5. Most of the rules are as expected; the two main novelties are that we replace ordinary substitution in the function application and case analysis rules with pattern substitution, and that there are some extra congruence rules to discard unneeded type annotations.

THEOREM 1 (Type Preservation). We have that:

- If $\cdot \vdash e : A$ and $e \mapsto^{PI} e'$, then $\cdot \vdash e' : A$. If $\cdot; \cdot \vdash u : A$ and $u \mapsto^{NI} u'$, then $\cdot; \cdot \vdash y' : A$. If $\cdot; \cdot \triangleright r : A$ and $r \mapsto^{PE} r'$, then $\cdot; \cdot \triangleright r' : A$. If $\cdot \triangleright t : A$ and $t \mapsto^{NE} t'$, then $\cdot \triangleright t' : A$.

The current form of the type theory has a preservation property, but no progress property. Progress fails because we have the $\perp L$ rule, which gives us a well-typed expression which can get stuck. This lets us typecheck programs with incomplete pattern-matching. without internalizing failure in the dynamic semantics of our language (e.g., with exceptions for match failure).

In the next section, we will give rules for judging when a pattern is complete, and prove their correctness relative to the current semantics. Then, once we modify the typing rules to include these rules, we can ensure that the progress property will hold.

2.1 From ML patterns to Focused Patterns

In this subsection, we will informally explain how to explain ML pattern matching in terms of focused patterns. For space reasons, we will not completely formalize this translation - we will merely indicate how this translation can be done. Let us return to the example from the introduction:

```
case e of
| (Inl x, Inl y) -> e1
| (Inr u, Inr v) -> e2
                 -> e3
1
```

To explain this construction, we need to understand two things: first, how to translate individual branches of a case statement using ML patterns into focused patterns, and second, how to combine them in a way that respects the sequential ordering of ML pattern matching.

Let us take the grammar of ML patterns to be

$$p_m$$
 ::= x \mid $\langle
angle$ \mid _ \mid (p_m, p'_m) \mid inl p_m \mid inr p_m

The main question in translation is how to treat the constructor patterns in p_m and inr p_m . Our solution is to view each of them as a focused pattern which uses \perp for either the left or right branch:

A	::= 	$\begin{array}{cccccccc} 1 & & A \times A & & A \to A \\ 0 & & A + A \end{array}$	Types
e	::=	$\langle \rangle \mid \langle e_1, e_2 \rangle$ inl $e \mid inr e$	Positive Intros
		a	I OSITIVE IIITOS
u	::=	$\lambda p. \ u \ \mid \ r$	Negative Intros
r	::= 	$ \begin{bmatrix} & [r_1 \mid r_2] & & e & & t \\ case(t, p \Rightarrow r) & & \bot & & r_1 \lor r_2 \\ \end{bmatrix} $	Positive Elims
t	::=	$x \hspace{.1 in} \hspace{.1 in} (e:A) \hspace{.1 in} \hspace{.1 in} t \hspace{.1 in} e$	Negative Elims
p	::= 	$\begin{array}{c c c c c c c c c c c c c c c c c c c $	Patterns
v	::= 	$egin{array}{c c c c c c c c c c c c c c c c c c c $	Values
Г	::=	$\cdot \mid \Gamma, x: A$	Variable Contexts
Δ	::=	$\cdot \mid \Delta, p: A$	Pattern Contexts



$$\begin{bmatrix} _ \end{bmatrix} = \top \\ \llbracket \langle \rangle \rrbracket = \langle \rangle \\ \llbracket x \rrbracket = x \\ \llbracket (p_m, p'_m) \rrbracket = \langle \llbracket p_m \rrbracket, \llbracket p'_m \rrbracket \rangle \\ \llbracket \operatorname{inl} p_m \rrbracket = [\operatorname{inl} \llbracket p_m \rrbracket | \operatorname{inr} \bot \\ \llbracket \operatorname{inr} p_m \rrbracket = [\operatorname{inl} \bot | \operatorname{inr} \llbracket p_m \rrbracket]$$

In the case of our example, we will end up with the three patterns $p_1 = \langle [inl \ x \ | \ inr \ \bot], [inl \ y \ | \ inr \ \bot] \rangle, p_2 = \langle [inl \ \bot \ | \ inr \ u] \rangle$ $[inl \perp | inr v]$, and $p_3 = \top$. Now, we must consider how to combine them, since the case form of the focused language requires a single pattern in the expression $case(t, p \Rightarrow r)$.

One way we might consider combining them is by using disjunction to construct $p_1 \vee p_2 \vee p_3$. However, this is not quite sufficient - the final wildcard pattern will not match any values of the form (inl v, inl v') or (inr v, inr v'), because the first two lines take priority. In order to express the correct behavior, we need is some way to say " p_3 , but not p_1 or p_2 ".

In Figure 6, we give a syntactic negation function on patterns. Given a pattern p, not(p) will return a pattern that fails to match on any value p matches on, and will matches on any value p fails on. The reason this is possible is because we included the failure \perp and nondeterministic choice $p_1 \vee p_2$ patterns. This turns out to be essential in the case of pairs; the negation $not(\langle p, p' \rangle) =$ $(\operatorname{not}(p), \top) \lor \langle \top, \operatorname{not}(p') \rangle$. Without the disjunction, there would be no pattern that could express this set of values.

Essentially, what \perp and $p \lor p'$ give us is an algebraic closure property for the language of patterns – at each type we have a boolean algebra consisting of the patterns of that type, ordered by the set inclusion on the set of values each pattern can match.

Using this, we can now give a focused pattern for this ML case expression, as $p_1 \vee (p_2 \wedge \mathsf{not}(p_1)) \vee (p_3 \wedge \mathsf{not}(p_1 \vee p_2))$. Furthermore, we can check for redundant patterns by testing to see whether each pattern plus the negation of its predecessors is empty or not. Using this basic structure we can construct a compositional translation of ML case expressions into the focused pattern calculus - even features like or-patterns are unproblematic,

$$\begin{array}{c} \overline{\Gamma \vdash \langle \rangle : 1} \ \mathrm{IR} & \frac{\Gamma \vdash e_1 : A_1 \quad \Gamma \vdash e_2 : A_2}{\Gamma \vdash \langle e_1, e_2 \rangle : A_1 \times A_2} \times \mathrm{R} \\ \overline{\Gamma \vdash e : A_1} \\ \overline{\Gamma \vdash e : A_1} \\ \overline{\Gamma \vdash e : A_1 + A_2} + \mathrm{L1} & \frac{\Gamma \vdash e : A_2}{\Gamma \vdash \operatorname{inr} e : A_1 + A_2} + \mathrm{L2} \\ \\ \overline{\Gamma \vdash u : A} \\ \overline{\Gamma \vdash u : A} \\ \mathrm{BLURR} \\ \hline \\ \overline{\Gamma; \Delta \vdash \lambda p. u : A \to B} \\ \overline{\Gamma; \Delta \vdash r : A} \\ \mathrm{BLURL} \\ \hline \\ \\ \overline{\Gamma; \Delta \vdash \lambda p. u : A \to B} \\ \overline{\Gamma; \langle D \mid D \rangle : A \to B} \\ \overline{\Gamma; \langle D \mid D \rangle : A \to B} \\ \overline{\Gamma; \langle D \mid D \rangle : A \to B} \\ \overline{\Gamma; \langle D \mid D \rangle : A \to A + A_2, \Delta \triangleright r : B} \\ \Gamma; \overline{\Gamma; \Gamma \mid D_1 \mid \operatorname{inr} p_2 : A_1 + A_2, \Delta \triangleright r : B} \\ \overline{\Gamma; \Gamma \mid D_1 \mid \operatorname{inr} p_2 : A_1 + A_2, \Delta \triangleright r : B} \\ \overline{\Gamma; \Gamma \mid A, \Delta \triangleright r : B} \\ \overline{\Gamma; r_1 \land A \triangleright r : B} \\ \overline{\Gamma; r_1 \land A \triangleright r : B} \\ \overline{\Gamma; r_1 \land A \triangleright r : B} \\ \overline{\Gamma; T \mid A, \Delta \triangleright r : B} \\ \overline{\Gamma; T \mid A, \Delta \triangleright r : B} \\ \overline{\Gamma; T \mid A, \Delta \triangleright r : B} \\ \overline{\Gamma; T \mid A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \land A \triangleright P : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \triangleright r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A, \Delta \vdash r : B} \\ \overline{\Gamma; P_1 \lor P_2 : A \lor P_1 \lor P_2 : A \vdash P_2 \lor A} \\ \overline{\Gamma; P_1 \lor P_2 : A \lor P_1 \lor P_2 \lor A \land P_1 \lor P_2 \lor A \land A \vdash E} \\ \overline{\Gamma; P_1 \lor P_2 \lor P_2 \lor A \land P_1 \lor P_2 \lor A \lor A \vdash E}$$

$ \begin{array}{l} [e/x]_A^{PI} \left< \right> \\ [e/x]_A^{PI} \left< e_1, e_2 \right> \\ [e/x]_A^{PI} \inf e \\ [e/x]_A^{PI} \inf e \\ [e/x]_A^{PI} u \end{array} $		$ \begin{array}{l} \langle \rangle \\ \langle [e/x]_A^{Pl} \; e_1, [e/x]_A^{Pl} \; e_2 \rangle \\ \operatorname{inl} \; [e/x]_A^{Pl} \; e \\ \operatorname{inr} \; [e/x]_A^{Pl} \; e \\ [e/x]_A^{Nl} \; u \end{array} $
$[e/x]^{\operatorname{Ni}}_A \lambda p. \ u \ [e/x]^{\operatorname{Ni}}_A r$	=	$ \begin{array}{l} \lambda p. \; [e/x]_A^{NI} \; u \\ [e/x]_A^{PE} \; r \end{array} $
$ \begin{array}{l} [e/x]_A^{PE} & []\\ [e/x]_A^{PE} & [r_1 \mid r_2] \\ [e/x]_A^{PE} & e \\ [e/x]_A^{PE} & t \\ [e/x]_A^{PE} & case(t, p \Rightarrow r) \\ [e/x]_A^{PE} & \bot \\ [e/x]_A^{PE} & r_1 \lor r_2 \end{array} $		$ \begin{bmatrix} [[e/x]_A^{PE} r_1 \mid [e/x]_A^{PE} r_2] \\ [e/x]_A^{PI} e \\ [e/x]_A^{NE} t \\ case([e/x]_A^{NE} t, p \Rightarrow [e/x]_A^{PE} r) \\ \bot \\ [e/x]_A^{PE} r_1 \lor [e/x]_A^{PE} r_2 \end{bmatrix} $
$\begin{array}{l} [e/x]_A^{NE} x\\ [e/x]_A^{NE} y\\ [e/x]_A^{NE} t e\\ [e/x]_A^{NE} (e:A') \end{array}$	= = =	$(e: A) y ([e/x]_A^{NE} t) ([e/x]_A^{Pl} e) ([e/x]_A^{Pl} e: A')$

Figure 3. Type Aware Substitution

$$\begin{split} \frac{\langle \langle v/p \rangle \rangle_A^{NI} u \rightsquigarrow u'}{\langle \langle v/p \rangle \rangle_A^{NI} \lambda p'. u \rightsquigarrow \lambda p'. u'} & \frac{\langle \langle v/p \rangle \rangle_A^{PE} r \rightsquigarrow r'}{\langle \langle v/p \rangle \rangle_A^{NI} r \rightsquigarrow r'} \\ \overline{\langle \langle v/p \rangle \rangle_A^{NI} \lambda p'. u \rightsquigarrow \lambda p'. u'} & \frac{\langle \langle v/p \rangle \rangle_A^{PE} r \rightsquigarrow r'}{\langle \langle v/p \rangle \rangle_A^{PE} r \rightsquigarrow r'} \\ \overline{\langle \langle v/p \rangle \rangle_A^{PE} r \rightsquigarrow r'} & \frac{\langle \langle v_2/p_2 \rangle \rangle_{A_2}^{PE} r' \rightsquigarrow r''}{\langle \langle v_1, v_2 \rangle / \langle p_1, p_2 \rangle \rangle_{A_1 \times A_2}^{PE} r \rightsquigarrow r''} \\ \frac{\langle \langle v/p_1 \rangle \rangle_{A_1}^{PE} r_1 \rightsquigarrow r'}{\langle \langle \inf v/[\inf p_1 | \inf p_2] \rangle \rangle_{A_1 + A_2}^{PE} [r_1 | r_2] \rightsquigarrow r'} \\ \frac{\langle \langle v/p_2 \rangle \rangle_{A_2}^{PE} r_2 \rightsquigarrow r'}{\langle \langle \inf v/[\inf p_1 | \inf p_2] \rangle \rangle_{A_1 + A_2}^{PE} [r_1 | r_2] \rightsquigarrow r'} \\ \overline{\langle \langle v/p_1 \rangle \rangle_A^{PE} r \rightsquigarrow r'} & \frac{\langle \langle v/p_2 \rangle \rangle_{A_1}^{PE} r \rightsquigarrow r'}{\langle \langle v/p_1 \rangle \rangle_A^{PE} r \rightsquigarrow r''} \\ \frac{\langle \langle v/p_1 \rangle \rangle_A^{PE} r \sim r'}{\langle \langle v/p_1 \wedge p_2 \rangle \rangle_A^{PE} r \sim r''} \\ \frac{\langle \langle v/p_1 \rangle \rangle_A^{PE} r_1 \lor r_2 \rightsquigarrow r'}{\langle \langle v/p_1 \vee p_2 \rangle \rangle_A^{PE} r_1 \lor r_2 \rightsquigarrow r'} \end{split}$$

Figure 4. Pattern Substitution

$$\begin{split} \frac{e_1 \ \mapsto^{PI} e_1'}{\langle e_1, e_2 \rangle \ \mapsto^{PI} \langle e_1', e_2 \rangle} & \frac{e_2 \ \mapsto^{PI} e_2'}{\langle v_1, e_2 \rangle \ \mapsto^{PI} \langle v_1, e_2' \rangle} \\ \frac{e \ \mapsto^{PI} e'}{\mathsf{inl} \ e \ \mapsto^{PI} \mathsf{inl} \ e'} & \frac{e \ \mapsto^{PI} e'}{\mathsf{inr} \ e \ \mapsto^{PI} \mathsf{inr} \ e'} & \overline{(e:A) \ \mapsto^{PI} e} \\ \frac{u \ \mapsto^{NI} u'}{u \ \mapsto^{PI} u'} & \frac{r \ \mapsto^{PE} r'}{r \ \mapsto^{NI} r'} & \overline{(u:A) \ \mapsto^{NI} u} \\ & \frac{e \ \mapsto^{PI} e'}{e \ \mapsto^{PE} e'} & \frac{t \ \mapsto^{NE} t'}{t \ \mapsto^{PE} t'} \\ & \frac{t \ \mapsto^{NE} t'}{\mathsf{case}(v;A), p \ \Rightarrow r) \ \mapsto^{PE} \mathsf{case}(t', p \ \Rightarrow r)} \\ & \frac{\langle \langle v/p \rangle \rangle_A^{PE} r \ \rightsquigarrow^{r'}}{\mathsf{case}(v:A), p \ \Rightarrow r) \ \mapsto^{PE} r'} & \frac{e \ \mapsto^{PE} e'}{(e:A) \ \mapsto^{PE} r} \\ & \frac{t \ \mapsto^{NE} t'}{\mathsf{t} \ e \ \mapsto^{PE} t'} & \frac{e \ \mapsto^{PE} e'}{(e:A) \ \mapsto^{PE} r} \\ & \frac{t \ \mapsto^{NE} t'}{\mathsf{case}(v;A), p \ \Rightarrow r) \ \mapsto^{PE} r'} & \frac{e \ \mapsto^{PE} e'}{(e:A) \ \mapsto^{NE} (e':A)} \\ & \frac{e \ \mapsto^{PE} e'}{(\lambda p. u:A \ \Rightarrow B) \ e \ \mapsto^{NE} (\lambda p. u:A \ \Rightarrow B) \ e'} \\ & \frac{\langle \langle v/p \rangle \rangle_A^{NI} u \ \rightsquigarrow^{NE} (u':B)} \end{split}$$



since we can regard an ML or-pattern like $p_m \mid p'_m$ as $[\![p_m]\!] \lor$ $(\llbracket p'_m \rrbracket \land \mathsf{not}(\llbracket p_m \rrbracket)).$

2.2 Deep Operations

The pattern context in our calculus is an ordered context, and as a result inverting the rules of this calculus will only let us decompose patterns at the leftmost end of the context. However, we will find it essential for us to introduce and operate on pattern hypotheses in the middle of the context. So in this subsection we will state the technical machinery we need to prove that it is admissible to operate on patterns anywhere within a context, and not just at the leftmost end. (In terms of the ordinary typed lambda calculus, we are developing machinery to work with commuting conversions.)

Since modifying pattern hypotheses within a contexts means we are adding and removing case statements and disjunctive choices deep within a context, we will also have to restructure the corresponding lambda term. We will give "deep" variants of the introduction rules and inversion principles, and then we will show that there is a deep version of pattern substitution, that exchange is admissible in the pattern context, and that deep substitution commutes with itself and with exchange operations. With all this technical machinery, we can restructure the pattern context almost as easily as an ordinary variable context, and compute what the modified proof terms look like. The algorithms for doing so are given in Figure 9 in the appendix.

To save space, we will let the symbol \oplus range over \vee and $[\cdot|\cdot]$, and likewise let the symbol \otimes and range over \wedge and $\langle \cdot, \cdot \rangle$. We will also use the following relations:

- $R^{[\cdot|\cdot]}(A_1, A_2, A) \equiv A = A_1 + A_2,$
- $R^{\vee}(A_1, A_2, A) \equiv A_1 = A \wedge A_2 = A$,
- $R^{\langle \cdot, \cdot \rangle}(A_1, A_2, A) \equiv A = A_1 \times A_2$, and
- $R^{\wedge}(A_1, A_2, A) \equiv A_1 = A \wedge A_2 = A.$

LEMMA 3 (Deep Introductions). We have that:

- If $\Gamma; \Delta, p_1 : A_1, \Delta' \triangleright r_1 : B$ and $\Gamma; \Delta, p_2 : A_2, \Delta' \triangleright r_2 : B$, and $R^{\oplus}(A_1, A_2, A)$, then $\Gamma; \Delta, p_1 \oplus p_2 : A, \Delta' \rhd \mathsf{Join}^{\oplus}(\Delta; r_1; r_2) : B.$
- If $\Gamma; \Delta, p_1 : A_1, p_2 : A_2, \Delta' \triangleright r : B$ and $R^{\otimes}(A_1, A_2, A)$, then $\Gamma; \Delta, p_1 \otimes p_2 : A_2, \Delta' \rhd r : B.$
- If $\Gamma, x : A; \Delta, \Delta' \triangleright r : B$, then $\Gamma; \Delta, x : A, \Delta' \triangleright r : B$.
- If $\Gamma; \Delta, \Delta' \triangleright r : B$, then $\Gamma; \Delta, \top, \Delta' \triangleright r : B$.
- If $r = \operatorname{Zero}^{c}(\Delta)$ and $c : A = \bot : A$ or c : A = [] : 0 then $\Gamma; \Delta, c: A, \Delta' \triangleright r: B.$

LEMMA 4 (Deep Inversions). We have that:

- If $\Gamma; \Delta, p_1 \oplus p_2 : A, \Delta' \rhd r : B \text{ and } R^{\oplus}(A_1, A_2, A)$, then $\Gamma; \Delta, p_1 : A, \Delta' \rhd \mathsf{Out}_1^{\oplus}(\Delta; r) : B \text{ and}$ $\Gamma; \Delta, p_2 : A, \Delta' \rhd \mathsf{Out}_2^{\oplus}(\Delta; r) : B.$
- If Γ ; Δ , $p_1 \otimes p_2 : A, \Delta' \rhd r : B$ and $R^{\otimes}(A_1, A_2, A)$, then Γ ; Δ , $p_1 : A_1, p_2 : A_2, \Delta' \rhd r : B$ and If Γ ; Δ , $x : A, \Delta' \rhd r : B$ then Γ , x : A; Δ , $\Delta' \rhd r : B$.
- If $\Gamma; \Delta, \top : A, \Delta' \triangleright r : B$ then $\Gamma; \Delta, \Delta' \triangleright r : B$.
- If Γ ; Δ , c : A, $\Delta' \triangleright r : B$ and $c : A = \bot : A$ or c : A = [] : 0, then $r = \operatorname{Zero}^{c}(\Delta)$.

LEMMA 5 (Operator Equalities). Assuming appropriate typings, we have that:

- $\operatorname{Out}_i^{\oplus}(\Delta; \operatorname{Join}^{\oplus}(\Delta; r_1; r_2)) = r_i$
- $\mathsf{Join}^{\oplus}(\Delta; \mathsf{Out}_1^{\oplus}(\Delta; r); \mathsf{Out}_2^{\oplus}(\Delta; r)) = r$
- $\operatorname{Out}_{i}^{\oplus}(\Delta; \operatorname{Out}_{i}^{\oplus'}(\Delta, p_{1} \oplus p_{2} : A, \Delta'; r)) =$ $\operatorname{Out}_{i}^{\oplus'}(\Delta, p_{i}: A_{i}, \Delta'; \operatorname{Out}_{i}^{\oplus}(\Delta; r))$
- $\operatorname{Out}_i^{\oplus}(\Delta; \operatorname{Join}^{\oplus'}(\Delta, p_1 \oplus p_2 : A, \Delta'; r_1; r_2)) =$ $\mathsf{Join}^{\oplus'}(\Delta, p_i : A_i, \Delta; \mathsf{Out}_i^{\oplus}(\Delta; r_1); \mathsf{Out}_i^{\oplus}(\Delta; r_2))$
- $\operatorname{Out}_i^{\oplus}(\Delta, p_1 \oplus' p_2 : A, \Delta'; \operatorname{Join}^{\oplus'}(\Delta; r_1; r_2)) =$ $\mathsf{Join}^{\oplus'}(\Delta; \mathsf{Out}_i^{\oplus}(\Delta, p_1 : A_1, \Delta'; r_1); \mathsf{Out}_i^{\oplus}(\Delta, p_2 : A_2, \Delta'; r_2))$

Given these deep operations, we can give a deep version of the pattern substitution principle:

PROPOSITION 1 (Deep Pattern Substitution). If $\cdot \vdash v : A$, then

- if $\Gamma; \Delta, p : A, \Delta' \vdash u : B$ and ${}^{\Delta}\langle\!\langle v/p \rangle\!\rangle_A^{NI} u \rightsquigarrow u'$, then we have $\Gamma; \Delta, \Delta' \vdash u' : B$.
- if $\Gamma; \Delta, p: A, \Delta' \triangleright r : B$ and $\Delta \langle \langle v/p \rangle \rangle_A^{PE} r \rightsquigarrow r'$, then we have $\Gamma; \Delta, \Delta' \rhd r' : B$.

We can also show that deep pattern substitutions commute:

LEMMA 6 (Pattern Substitution Reordering). If we have that $\cdot \vdash$ $\begin{array}{l} \text{Definition for contrastic producting } f \quad \forall e \text{ nuter that } \\ v: A, \cdot \vdash v': B, \text{ and } \Gamma; \Delta, p: A, \Delta', p': B, \Delta'' \triangleright r: C, \text{ then:} \\ \\ \overset{\Delta}{\langle \langle v/p \rangle \rangle}_{A}^{PE} r \sim r_{1} \text{ and } \\ \overset{\Delta, \Delta'}{\langle \langle v'/p' \rangle \rangle}_{B}^{PE} r_{1} \sim r_{2}, \\ \text{ if and only if } \\ \\ \overset{\Delta, p: A, \Delta'}{\langle \langle v'/p' \rangle \rangle}_{A}^{PE} r \sim r_{1}' \text{ and } \\ \overset{\Delta}{\langle \langle v'/p' \rangle \rangle}_{B}^{PE} r_{1}' \sim r_{2}, \end{array}$

Finally, we can show that exchange is an admissible property of

the pattern context:

PROPOSITION 2 (Exchange). We can show that $\textit{if } \Gamma; \Delta, p: A, \Delta', \Delta'' \vartriangleright r: B$ then $\Gamma; \Delta, \Delta', p : A, \Delta'' \triangleright \mathsf{Ex}(\Delta; p : A; \Delta'; r) : B$ LEMMA 7 (Substitution/Exchange Reordering). We have that:

• If we have that: $\begin{array}{l} \bullet \cdot \vdash v : A, \\ \bullet \ ^{\Delta} \langle \langle v/p \rangle \rangle_{A}^{PE} \ r \rightsquigarrow r', and \\ \bullet \ \Gamma; \Delta, p : A, \Delta', p' : B, \Delta'', \Delta''' \rhd r : C \end{array}$ $\begin{array}{l} & (1, \underline{-}, p + 1), \underline{-}, p + 2, \underline{-}, \underline{-} \neq 1 + 0 \\ & \text{then it follows that} \\ & \Delta \langle\!\langle v/p \rangle\!\rangle_A^{PE} \; \mathsf{Ex}(\Delta, p : A, \Delta'; \; p' : B; \; \Delta''; \; r) \! \rightsquigarrow \! r' \\ & \text{where } r' = \mathsf{Ex}(\Delta, \Delta'; \; p' : B; \; \Delta''; \; r) \end{array}$ • If we have that: • $\cdot \vdash v : B$, • $\Gamma; \Delta, p: A, \Delta', \Delta'', p': B, \Delta''' \triangleright r: C$ • $\Delta, p: A, \Delta', \Delta'' \langle \langle v/p' \rangle \rangle_B^{PE} r \sim r'$ then it follows that $\overset{\Delta,\Delta',p:A,\Delta''}{\longrightarrow} \langle \langle v/p' \rangle \rangle_B^{PE} \operatorname{Ex}(\Delta; \ p:A; \ \Delta'; \ r) \rightsquigarrow r''$ where $r'' = \operatorname{Ex}(\Delta; \ p:A; \ \Delta'; \ r')$ • If we have that: If we have made • $\vdash v : A,$ • $\Delta \langle \langle v/p \rangle \rangle_A^{PE} r \sim r', and$ • $\Gamma; \Delta, p : A, \Delta', \Delta'' \succ r : B,$ then it follows that $\Delta, \Delta' \langle \langle v/p \rangle \rangle_A^{PE} \mathsf{Ex}(\Delta; p : A; \Delta'; r) \sim r'$ • If we have that: • $\cdot \vdash v : B$, • $\stackrel{\Delta,p:A,\Delta'}{\longrightarrow} \langle \langle v/p \rangle \rangle_B^{PE} r \rightsquigarrow r', and$ • $\Gamma; \Delta, p: A, \Delta', p': B, \Delta'', \Delta''' \triangleright r: C$ $\begin{array}{l} \text{then it follows that:} \\ {}^{\Delta,\Delta'} \langle \langle v/p' \rangle \rangle_B^{PE} \; \mathsf{Ex}(\Delta; \; p:A; \; \Delta',p':B,\Delta''; \; r) \rightsquigarrow r'' \\ \end{array}$ where $r'' = \mathsf{Ex}(\Delta; p:A; \Delta', \Delta''; r')$

3. Coverage Checking

Pattern substitution is a partial relation, and this means that a given pattern may not have any values it will match - for example $[\operatorname{inl} x | \operatorname{inr} \bot] \land [\operatorname{inl} \bot | \operatorname{inr} y] - \operatorname{or} \operatorname{it} \operatorname{may}$ be able to match in multiple ways – for example $x \vee \langle \top, y \rangle$. It would be very useful to characterize patterns that have exactly one possible match for each value.

To do this, we introduce two judgements. The first, $p \det A$, checks for determinacy – if $p \det A$ is derivable, then p matches each value of type A at most once. The second, p covers A, checks whether a pattern covers all possible values. That is, if p covers Ais derivable, then p matches every value in A in at least one way. Both of these judgements are given in Figure 7. Each one inductively follows the structure of p, until we reaches a disjunctive choice $p \vee p'$.

In both cases, this is somewhat problematic, because whether a disjunctive pattern $p \lor p'$ is deterministic depends on the *interaction* between the two branches. For example, if $p = [inl \ x \mid inr \perp]$ and $p' = [\text{inl} \perp | \text{inr } y]$, then neither p nor p' is a covering pattern, but their disjunctive union is. Conversely, both p = x and $p' = \top$ are deterministic patterns, but $x \vee \langle \rangle$ is nondeterministic.

So we need to check that the intersection of p and p' is empty to claim that $p \lor p'$ det A. Likewise we need to ensure that there are no values that both p and p' fail to match in order to claim that $p \lor p'$ covers A – that is, we need to ensure that the intersection of the complements is empty.

To do this, we introduce a third judgement, p_1, \dots, p_n fail A. The existence of a derivation of this judgement will imply that for every value of type A, there is some p_i such that the match will fail. Then, for determinacy we can model the emptiness of the intersections with p, p' fail A, and for coverage we can model the emptiness of the intersection of the complements with not(p), not(p') fail A.

First, we show that our syntactic negation operation is a genuine complement:

PROPOSITION 3 (Negation). If we have derivations:

- $\Gamma; \Delta_1, p: A, \Delta'_1 \triangleright r_1 : C,$
- $\Gamma; \Delta_2, \operatorname{not}(p) : A, \Delta'_2 \triangleright r : C,$
- and $\Gamma \vdash v : A$,

then

1. either ${}^{\Delta_1}\langle\!\langle v/p \rangle\!\rangle_A^{PE} r_1 \rightsquigarrow r' \text{ or } {}^{\Delta_2}\langle\!\langle v/p \rangle\!\rangle_A^{PE} r_2 \rightsquigarrow r'$ 2. it is not the case that both ${}^{\Delta_1}\langle\!\langle v/p \rangle\!\rangle_A^{PE} r_1 \rightsquigarrow r'$ and ${}^{\Delta_2}\langle\!\langle v/p \rangle\!\rangle_A^{PE} r_2 \rightsquigarrow r''$

Next, we prove the soundness of the judgements we have described.

LEMMA 8 (Failure). If we have derivations

- p_1, \cdots, p_n fail A
- $\Gamma; \Delta, p_1: A, \cdots, p_n: A, \Delta' \triangleright r_1: C$
- $\Gamma \vdash v : A$

then it is not the case that there exist r_2, \dots, r_n such that for all $i \in \{1 \dots n\}, {}^{\Delta} \langle \langle v/p_i \rangle \rangle_A^{PE} r_i \sim r_{i+1}.$

THEOREM 2 (Determinacy). If we have derivations

• $\Gamma; \Delta, p: A, \Delta' \triangleright r: C$ • $\cdot \vdash v : A$ • $p \det A$ • $D :: {}^{\Delta} \langle\!\langle v/p \rangle\!\rangle_A^{PE} r \rightsquigarrow r'$

then if $D' :: {}^{\Delta} \langle\!\langle v/p \rangle\!\rangle_{A}^{PE} r \rightsquigarrow r''$, then D = D'.

Here, we use D and D' to name the particular derivations of the pattern substitution. So if $p \det A$, then for any derivation that includes p in its pattern context, any two pattern substitutions for p must be identical.

For coverage, we assert that for any derivation with p in its context, then if p covers A then there must exist a pattern substitution.

THEOREM 3 (Coverage). If we have derivations

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

then ${}^{\Delta}\langle\!\langle v/p \rangle\!\rangle_{\scriptscriptstyle A}^{PE} r \rightsquigarrow r'$.

Armed with our coverage judgement, we can modify the two rules introducing pattern binders to require that they are deterministic and cover all the possibilities:

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

$$\frac{p \text{ covers } A \quad p \text{ det } A \quad \Gamma \rhd t: A \quad \Gamma; p: A \rhd r: B}{\Gamma; \cdot \rhd \mathsf{case}(t, p \Rightarrow r): B}$$

With our modified rules, it is now possible to give a progress theorem for this calculus:

THEOREM 4 (Progress). With the modified rules, we have that:

- If $\cdot \vdash e : A$, then $e \mapsto^{PI} e'$ or e = v.
- If $:: \vdash u : A$, then $u \mapsto^{NI} u'$ or u = v. If $:: \vdash r : A$, then $r \mapsto^{PE} r'$ or r = v.

I Iguie of I attern i tegation	Figure	6.	Pattern	Negatior
--------------------------------	--------	----	---------	----------

• If
$$\cdot \rhd t : A$$
, then $t \mapsto^{NE} t'$ or $t = (v : A)$.

Pattern Compilation 4.

In this section, we give an algorithm that accepts a deterministic and covering set of patterns, and produces a set of patterns that 1) make no use of \perp or \vee in them, and 2) only use conjunctive patterns of the form $x \wedge p$. The first restriction syntactically guarantees that matching will never fail or backtrack, and the second restriction guarantees that no value need ever be tested twice (for instance, by $\langle p, p' \rangle \land \langle p, p' \rangle.$

$$\begin{array}{cccc} c & ::= & [] & | & [\mathsf{inl} \ c_1 \ | \ \mathsf{inr} \ c_2] & | & x \land [\mathsf{inl} \ c_1 \ | \ \mathsf{inr} \ c_2] & | & x \\ & | & \langle \rangle & | & \langle c_1, c_2 \rangle & | & x \land \langle c_1, c_2 \rangle & | & \top \end{array}$$

Matches against patterns of this form are easily implemented with simple nested case statements and let-bindings. So our goal is to transform a deterministic, covering pattern p (with body r), into an equivalent form pattern c (with body r'), such that any match of that value against c in r' gives the same result as matching that value against p in r'.

We give the core pattern compilation algorithm in Figure 8. In order to strengthen the induction hypothesis enough, we cannot operate on a single derivation $\Gamma; p: A, \Delta \triangleright r : B$. Instead, we have to generalize the induction hypothesis along two dimensions.

First, we strengthen our induction hypothesis so that it considers an entire sequence of patterns $\Gamma; p_1 : A, \cdots, p_n : A_n, \Delta \triangleright r :$ B. This will let us decompose conjunctive and pair patterns before applying our induction hypothesis (this is the function of the $\mathsf{Decompose}(p:A)$ function).

Then, instead of considering a single derivation, we consider a whole set S of them, with each row of the form $\Gamma; q_1 : A, \dots, q_n : A_n, \Delta \triangleright$ r: B, where q_i is either \top or p_i . This generalization helps us compile disjunctive patterns.

Concretely, imagine we have a row of the form Γ ; $p_1 \vee p'_1 : A, \Delta \triangleright$ $r \lor r' : B$. By inversion we can get two subderivations

- $\Gamma; p_1 : A, \Delta \triangleright r : B$
- $\Gamma; p'_1 : A, \Delta \triangleright r' : B$

Notice that we cannot directly apply our induction hypothesis, because these two subderivations do not have the same row, and so we cannot collect them in the same set S. However, we can introduce some harmless \top patterns to get

- $\Gamma; p_1: A, \top : A, \Delta \triangleright r : B$
- $\Gamma; \top : A, p'_1 : A, \Delta \triangleright r' : B$

Now, both of these match the form of our generalization, and we can collect them in the same set. This is why $\mathsf{Decompose}(p \lor p' : A) =$ p:A,p':A.

The pattern compiler needs two more supporting definitions. The first, And_A(Δ ; c_1 ; c_2 ; r) (given in Figure 13), takes two

$$\frac{p_1 \text{ covers } A}{\langle p_1, p_2 \rangle \text{ covers } A \times B}$$

x

 $not(p_1), not(p_2)$ fail A p_1 covers A p_2 covers A $p_1 \lor p_2$ covers A $p_1 \wedge p_2$ covers A

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

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

$$\frac{\overrightarrow{p}, \overrightarrow{p}' \text{ fail } A}{\overrightarrow{p}, x, \overrightarrow{p}' \text{ fail } A} \qquad \frac{\overrightarrow{p} \text{ fail } A_1}{\overrightarrow{p} \text{ fail } A} \qquad \frac{\overrightarrow{p} \text{ fail } A_1}{\overrightarrow{p} \text{ fail } A}$$

$$\frac{\overrightarrow{p} \text{ fail } A}{\overrightarrow{p}, x, \overrightarrow{p}' \text{ fail } A} \qquad \frac{\overrightarrow{p} \text{ fail } A_1}{\overrightarrow{p} \text{ fail } A_1 + A_2}$$

$$\frac{\overrightarrow{p} \text{ fail } A}{\overrightarrow{\langle p, p' \rangle} \text{ fail } A \times B} \qquad \frac{\overrightarrow{p}' \text{ fail } B}{\overrightarrow{\langle p, p' \rangle} \text{ fail } A \times B}$$



compiled patterns and produces a third compiled pattern c (and body r') that is equivalent to the pattern $c_1 \wedge c_2$. We need this function because our compiled patterns restrict how we can use conjunction in order to prohibit redundant re-tests of the same value.

LEMMA 9 (Conjunction Simplification). If Γ ; Δ , c_1 : A, c_2 : A, $\Delta' \triangleright$ $r: B and \cdot \vdash v: A$, then

- $(c; r') = \operatorname{And}_A(\Delta; c_1; c_2; r),$ $\Gamma; \Delta, c: A, \Delta' \rhd r': B, and$ $\Delta \langle\!\langle v/c \rangle\!\rangle_A^{PE} r' \rightsquigarrow r'' if and only if <math>\Delta \langle\!\langle v/c_1 \land c_2 \rangle\!\rangle_A^{PE} r \rightsquigarrow r''$

Finally, when we split a sum pattern in the third clause of the compilation function, we make two recursive calls to $Opt(\Delta; S)$. This gives us two rows of the form $c_1, \overrightarrow{cs_1}$ and $c_2, \overrightarrow{cs_2}$. We cannot assume these two rows have the same form, so we need a function to merge them and make them the same, so that we can use the +Lrule. This is what $Merge_A(\Delta_1; c_1; r_1; \Delta_2; c_2; r_2)$ does:

$$\begin{aligned} & \mathsf{Opt}(:\{\{(\cdot;r)\}\}) = \\ & (\cdot;r) \\ & \mathsf{Opt}([]:0,\overrightarrow{p}:\overrightarrow{A};S) = \\ & ([],\overrightarrow{\top}:\overrightarrow{A};[]) \\ & \mathsf{Opt}([\mathsf{inl}\ p_1 \mid \mathsf{inr}\ p_2]:A_1 + A_2, \overrightarrow{p}:\overrightarrow{A};S) = \\ & \mathsf{let}\ (c_1,\overrightarrow{cs_1};r_1) = \mathsf{Opt}(p_1:A_1;\overrightarrow{p}:\overrightarrow{A};\mathsf{Left}(S)) \\ & \mathsf{let}\ (c_2,\overrightarrow{cs_2};r_2) = \mathsf{Opt}(p_2:A_2;\overrightarrow{p}:\overrightarrow{A};\mathsf{Right}(S)) \\ & \mathsf{let}\ (\overrightarrow{cs};r_1';r_2') = \mathsf{Merge}_{\overrightarrow{A}}^*(c_1:A_1;\ \overrightarrow{cs_1};\ r_1;\ c_2:A_2;\ \overrightarrow{cs_2};\ r_2) \\ & ([\mathsf{inl}\ c_1 \mid \mathsf{inr}\ c_2];\ \overrightarrow{cs};\ [r_1 \mid r_2]) \\ & \mathsf{Opt}(p:A,\overrightarrow{p}:\overrightarrow{A};S) = \\ & \mathsf{let}\ (\overrightarrow{cs};r) = \mathsf{Opt}(\mathsf{Decompose}(p:A),\overrightarrow{p}:\overrightarrow{A};\mathsf{Split}(p;\ S)) \\ & \mathsf{Coalesce}_A(p;\ \overrightarrow{cs};\ r) \end{aligned}$$



LEMMA 10 (Pattern Merging). If we have that

• $\Gamma; \Delta_1, c_1 : A, \Delta'_1 \rhd r_1 : B$ • $\Gamma; \Delta_2, c_2 : A, \Delta'_2 \rhd r_2 : B$

- $\cdot \vdash v : A$

then

•
$$(c; r'_1; r'_2) = \text{Merge}_A(\Delta_1; c_1; r_1; \Delta_2; c_2; r_2)$$

- $\Gamma; \Delta_1, c: A, \Delta'_1 \triangleright r'_1 : B$

- Γ , Δ_1 , c : A, $\Delta_1 \succ r_1 \cdot D$ Γ ; Δ_2 , c : A, $\Delta'_2 \succ r'_2$: B• $\Delta \langle\!\langle v/c \rangle\!\rangle_A^{PE} r_1 \sim r''_1$ if and only if $\Delta \langle\!\langle v/c_1 \rangle\!\rangle_A^{PE} r_1 \sim r''_1$ $\Delta \langle\!\langle v/c \rangle\!\rangle_A^{PE} r'_2 \sim r''_2$ if and only if $\Delta \langle\!\langle v/c_2 \rangle\!\rangle_A^{PE} r_2 \sim r''_2$

At last, we can show the correctness of our scheme with the following two theorems:

THEOREM 5 (Soundness of Pattern Compilation). If

- $\Delta = p_1 : A_1, \cdots, p_n : A_n$,
- S is a set such that for every $(q_1, \dots, c_n; r) \in S$, $q_i \in \{p_i, \top\}$, and $\Gamma; q_1 : A_1, \dots, q_n : A_n, \Delta' \triangleright r : B$
- $(c_1, \cdots, c_n; r'_1) = \mathsf{Opt}(\Delta; S)$

then

- $\Gamma; c_1 : A_1, \cdots, c_n : A_n, \Delta' \triangleright r'_1 : B, and$ if there exists a unique $((q_1, \cdots, c_n; r_1) \in S, such that for all i \in \{1 \cdots n\}, \langle \langle v_i/q_i \rangle \rangle_{A_i}^{PE} r_i \rightsquigarrow r_{i+1} uniquely,$ then there exist $r'_2 \cdots r'_{n+1}$ such that for all $i \in \{1 \cdots n\}, \langle \langle v_i/c_i \rangle \rangle_{A_i}^{PE} r'_i \rightsquigarrow r_{i+1} and r'_{n+1} = r_{n+1}.$

THEOREM 6 (Termination of Pattern Compilation). If

- $\Delta = p_1 : A_1, \cdots, p_n : A_n$,
- S is a set such that for every $(q_1, \dots, c_n; r) \in S$, $q_i \in \{p_i, \top\}$,
- there exists a unique $((q_1, \dots, c_n; r_1) \in S, q_i \in \{p_i, +\},$ $there exists a unique ((q_1, \dots, c_n; r_1) \in S, such that for all <math>i \in \{1 \dots n\}, \langle\!\langle v_i/q_i \rangle\!\rangle_{A_i}^{PE} r_i \sim r_{i+1}$ uniquely,

then there is a $(c_1, \cdots, c_n; r'_1) = \mathsf{Opt}(\Delta; S).$

Now, note that any pattern for which we can derive $p \det A$ and p covers A is one which our compilation algorithm will successfully run on, since such a pattern will uniquely match any value, which is precisely the precondition of the compilation algorithm.

5. **Extensions and Future Work**

Extending this calculus to support parametric polymorphism and iso-recursive types is very easy. Since universal quantification is a negative type, like the function space, it does not affect the pattern language. However, existentials are positive, so in addition to an introduction form pack(A, e) at type $\exists \alpha. B$, we can also add a pattern elimination form $pack(\alpha, p)$. Likewise, the introduction form for a recursive type $\mu\alpha A$ will be a term fold e, with a corresponding pattern fold p.

The extensions needed to characterize features like GADTs [8], and dependent types [17, 12] are much more challenging. All of these systems use types to constrain the set of patterns that are needed for coverage in sophisticated ways, and requiring that unneeded patterns not be retained within the pattern.

Another direction would to examine how pattern matching should function in a call-by-name setting. To model such languages, we would have to interleave reduction and pattern substitution, so that pattern matching could force evaluation on an as-needed basis.

Related Work 6.

We were inspired to view pattern matching as arising from the invertible left rules of the sequent calculus due to the work of Kesner et al [9], and Cerrito and Kesner [4]. We have extended their work by building on a focused sequent calculus. This permits 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, we introduced the failure and nondeterministic choice patterns, which permit us to explain the sequential pattern matching found in functional languages, as well as to describe coverage checking and compilation.

Focusing was introduced by Andreoli [1], 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 [10] give calculi for focused intuitionistic logic, which they relate to both linear and classical logic. Neither of these have proof terms.

Our pattern substitution is a restricted form of hereditary substitution, which Watkins et al. [16] introduced as a way of reflecting the computational content of structural proofs of cut admissibility [14].

Girard's work on ludics [7] 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, at the (temporary!) cost of soundness and determinism, respectively.

Zeilberger [18] gives an analysis of focusing in terms of Dummett's notion of logical harmony [5], and uses this analysis to construct a theory which relates patterns to continuations. He does not give a coverage algorithm, but instead builds in coverage as a declarative requirement of his typing rules.

In real compilers, there are two classical approaches to compiling pattern matching, either by constructing a decision tree (described by Cardelli [3]) or building a backtracking automaton (described by Augustsson [2]). In our system, both of these approaches can be represented uniformly, since backtracking is cleanly isolated with the use of the nondeterministic disjunction pattern $p_1 \vee p_2$ and the abort pattern [].

$\begin{array}{l} Split(x;\;(x,\overrightarrow{qs};r))\\ Split(x;\;(\top,\overrightarrow{qs};r))\\ Split(\langle\rangle;\;(\langle\rangle,\overrightarrow{qs};r))\\ Split(\langle\rangle;\;(\top,\overrightarrow{qs};r))\\ Split(\langle p_1,p_2\rangle;\;(\langle p_1,p_2\rangle,\overrightarrow{qs};r))\\ Split(\langle p_1,p_2\rangle;\;(\top,\overrightarrow{qs};r)) \end{array}$	$ \{ (\vec{q}\vec{s};r) \} \\ \{ (\vec{q}\vec{s};r) \} \\ \{ (\vec{q}\vec{s};r) \} \\ \{ (\vec{q}\vec{s};r) \} \\ \{ (p_1,p_2,\vec{q}\vec{s};r) \} \\ \{ (\top,\top,\vec{q}\vec{s};r) \} $
$\begin{array}{l} Split(\top;\ (\top,\overrightarrow{qs};r))\\ Split(p_1 \wedge p_2;\ (p_1 \wedge p_2,\overrightarrow{qs};r))\\ Split(p_1 \wedge p_2;\ (\top,\overrightarrow{qs};r))\\ Split(\bot;\ (\top,\overrightarrow{qs};r))\\ Split(p_1 \vee p_2;\ (p_1 \vee p_2,\overrightarrow{qs};r_1 \vee r_2))\\ \end{array}$	$ \begin{array}{l} \{(\overrightarrow{qs};r)\} \\ \{(p_1,p_2,\overrightarrow{qs};r)\} \\ \{(\top,\top,\overrightarrow{qs};r)\} \\ \{(\overrightarrow{qs};r)\} \\ \{(p_1,\top,\overrightarrow{qs};r_1), \\ (\top,p_2,\overrightarrow{qs};r_2)\} \\ \{(\top,\top,\overrightarrow{qs};r)\} \end{array} $
Coalesce _A (\top ; \overrightarrow{cs} ; r) Coalesce _A ($p_1 \land p_2$; $c_1, c_2, \overrightarrow{cs}$; r) Coalesce _A (\perp ; \overrightarrow{cs} ; r) Coalesce _A ($p_1 \lor p_2$; $c_1, c_2, \overrightarrow{cs}$; r) Coalesce _A ($x; \overrightarrow{cs}; r$) Coalesce ₁ ($\langle \rangle; \overrightarrow{cs}; r$) Coalesce _{A × B} ($\langle p_1, p_2 \rangle; c_1, c_2, \overrightarrow{cs}; r$)	$\begin{array}{l} (x,\overrightarrow{cs};\ r)\\ And_A(\cdot;\ c_1;\ c_2;\ r)\\ (\top,\overrightarrow{cs};\ r)\\ And_A(\cdot;\ c_1;\ c_2;\ r)\\ (x,\overrightarrow{cs};\ r)\\ (x,\overrightarrow{cs};\ r)\\ ((c_1,c_2),\overrightarrow{cs};\ r)\end{array}$

Figure 11. Pattern Compilation: Supporting Definitions

Fessant and Maranget [6] describe a modern algorithm for pattern compilation which operates over matrices of patterns. Their algorithm tries to construct an efficient backtracking automaton, whereas our compilation algorithm is a reconstruction of the decision trees method. However, the rows of their matrices correspond, roughly, to the elements of the input set S in our compilation algorithm, and we believe that we have an unordered set because we work with *unbiased* choice.

Maranget [11] describes an algorithm for generating warnings for non-exhaustive matches and useless clauses, for both strict and lazy languages. This is difficult to compare with our coverage algorithm, since it operates on a substantially different principle. He does consider efficiency, a question we have largely ignored.

Sestoft [15] shows how to generate pattern matching code via partial evaluation. He does not consider the question of coverage checking.

A. Appendix Title

Acknowledgments

The author would like to thank Jonathan Aldrich, Robert Harper, Dan Licata, William Lovas, Frank Pfenning, Jason Reed, Kevin Watkins, and Noam Zeilberger for valuable 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. Andreoli. Logic Programming with Focusing Proofs in Linear Logic. Journal of Logic and Computation, 2(3):297, 1992.
- [2] L. Augustsson. Compiling pattern matching. Proc. of a conference on Functional programming languages and computer architecture table of contents, pages 368–381, 1985.
- [3] L. Cardelli. Compiling a functional language. In LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional

Left(\emptyset) Left({ $(\top, \overrightarrow{q}; r)$ } $\cup S$) Left({([inl $p_1 \mid inr p_2], \overrightarrow{q}; [r_1 \mid r_2])$ } $\cup S$)	=	
$\begin{split} & Right(\emptyset) \\ & Right(\{(\top, \overrightarrow{q}; r)\} \cup S) \\ & Right(\{([inl \ p_1 \mid inr \ p_2], \overrightarrow{q}; [r_1 \mid r_2])\} \cup S) \end{split}$	=	
$\begin{array}{l} Decompose(\top:A)\\ Decompose(p_1 \wedge p_2:A)\\ Decompose(\bot:A)\\ Decompose(p_1 \lor p_2:A)\\ Decompose(x:A)\\ Decompose(\langle \rangle:1)\\ Decompose(\langle p_1, p_2 \rangle:A \times B) \end{array}$		$p_1: A, p_2: A$ $p_1: A, p_2: A$ $p_1: A, p_2: B$

Figure 12. Pattern Compilation: More Supporting Definitions

programming, pages 208–217, New York, NY, USA, 1984. ACM Press.

- [4] S. Cerrito and D. Kesner. Pattern matching as cut elimination. *Theoretical computer science*, 323(1-3):71–127, 2004.
- [5] M. Dummett. The Logical Basis of Metaphysics. Duckworth, 1991.
- [6] F. L. Fessant and L. 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.
- [7] J. Girard. Locus Solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(03):301– 506, 2001.
- [8] S. 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.
- [9] D. Kesner, L. Puel, and V. Tannen. A Typed Pattern Calculus. Information and Computation, 124(1):32–61, 1996.
- [10] C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In 16th EACSL Annual Conference on Computer Science and Logic. Springer-Verlag, 2007.
- [11] L. Maranget. Warnings for pattern matching. Journal of Functional Programming, 2007.
- [12] C. McBride. Epigram. Types for Proofs and Programs, Torino, 2003, 3085:115–129, 2003.
- [13] R. Milner. The Definition of Standard Ml:(revised). MIT Press, 1997.
- [14] F. Pfenning. Structural Cut Elimination I. Intuitionistic and Classical Logic. *Information and Computation*, 157(1-2):84–141, 2000.
- [15] P. Sestoft. ML pattern match compilation and partial evaluation. Lecture Notes in Computer Science, 1110:446–??, 1996.
- [16] 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.
- [17] H. Xi. Dependently Typed Pattern Matching. Journal of Universal Computer Science, 9(8):851–872, 2003.
- [18] N. 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.

 $\mathsf{Out}_i^\oplus(\cdot; r_1 \oplus r_2)$ = r_i $\mathsf{Out}_i^{\oplus}(\top : A, \Delta; r)$ $\operatorname{Out}_i^{\oplus}(\Delta; r)$ = $\mathsf{Out}_i^{\oplus}(p_1 \wedge p_2 : A, \Delta; r)$ $\operatorname{Out}_{i}^{\oplus}(p_{1}: A, p_{2}: A, \Delta; r)$ = $\operatorname{Out}_i^{\oplus}(\bot:A,\Delta;\bot)$ = $\operatorname{Out}_{i}^{\oplus}(p_{1}:A,\Delta;r_{1}) \vee \operatorname{Out}_{i}^{\oplus}(p_{2}:B,\Delta;r_{2})$ $\mathsf{Out}_i^{\oplus}(p_1 \lor p_2 : A, \Delta; r_1 \lor r_2)$ = $\operatorname{Out}_{i}^{\oplus}(\Delta; r)$ $\operatorname{Out}_i^{\oplus}(\langle \rangle : 1, \Delta; r)$ = $\operatorname{Out}_{i}^{\oplus}(\langle p_1, p_2 \rangle : A \times B, \Delta; r)$ $\mathsf{Out}_i^{\oplus}(p_1:A,p_2:B,\Delta;r)$ = $\mathsf{Out}_i^{\oplus}([]:0,\Delta;[])$ = $\operatorname{Out}_{i}^{\oplus}([\operatorname{inl} p_{1} \mid \operatorname{inr} p_{2}] : A + B, \Delta; [r_{1} \mid r_{2}])$ = $\left[\mathsf{Out}_i^{\oplus}(p_1:A,\Delta; r_1) \mid \mathsf{Out}_i^{\oplus}(p_2:B,\Delta; r_2)\right]$ $\mathsf{Join}^{\oplus}(\cdot; r_1; r_2)$ = $r_1 \oplus r_2$ $\mathsf{Join}^{\oplus}(\top : A, \Delta; r_1; r_2)$ $\mathsf{Join}^{\oplus}(\Delta; r_1; r_2)$ = $\mathsf{Join}^{\oplus}(p_1 \wedge p_2 : A, \Delta; r_1; r_2)$ $\mathsf{Join}^{\oplus}(p_1:A,p_2:A,\Delta;\ r_1;\ r_2)$ = $\mathsf{Join}^{\oplus}(\bot:A,\Delta;\ \bot;\ \bot)$ = $\mathsf{Join}^{\oplus}(p_1 \vee p_2 : A, \Delta; r_1 \vee r_2; r'_1 \vee r'_2)$ $\mathsf{Join}^{\oplus}(p_1:A,\Delta; r_1; r'_1) \lor \mathsf{Join}^{\oplus}(p_2:B,\Delta; r_2; r'_2)$ = $\mathsf{Join}^{\oplus}(\langle\rangle:1,\Delta;\ r_1;\ r_2)$ $\mathsf{Join}^{\oplus}(\Delta; r_1; r_2)$ = $\mathsf{Join}^{\oplus}(\ddot{\langle}p_1,p_2\rangle:A\times B,\Delta;\ r_1;\ r_2)$ $\mathsf{Join}^{\oplus}(p_1:A,p_2:B,\Delta;\ r_1;\ r_2)$ = $\mathsf{Join}^{\oplus}([]:0,\Delta;[];[])$ = Π $\left[\mathsf{Join}^{\oplus}(p_1:A,\Delta; r_1; r_1') \mid \mathsf{Join}^{\oplus}(p_2:B,\Delta; r_2; r_2')\right]$ $\mathsf{Join}^{\oplus}([\mathsf{inl} \ p_1 \ | \ \mathsf{inr} \ p_2] : A + B, \Delta; \ [r_1 \ | \ r_2]; \ [r'_1 \ | \ r'_2])$ = $Zero^{c}(\cdot)$ = c $\operatorname{Zero}^{c}(\dot{\top}:A,\Delta)$ $Zero^{c}(\Delta)$ = $\mathsf{Zero}^c(p_1 \wedge p_2 : A, \Delta)$ $\mathsf{Zero}^c(p_1:A,p_2:A,\Delta)$ = $\operatorname{\mathsf{Zero}}^c(\bot:A,\Delta)$ = $\mathsf{Zero}^c(p_1 \lor p_2 : A, \Delta)$ $\operatorname{\mathsf{Zero}}^c(p_1:A,\Delta) \vee \operatorname{\mathsf{Zero}}^c(p_2:A,\Delta)$ = $\operatorname{Zero}^{c}(x:A,\Delta)$ = $Zero^{c}(\Delta)$ $\operatorname{Zero}^{c}(\langle \rangle : 1, \Delta)$ = $Zero^{c}(\Delta)$ $\operatorname{\mathsf{Zero}}^c(\langle p_1, p_2 \rangle : A \times B, \Delta)$ $\operatorname{\mathsf{Zero}}^c(p_1:A,p_2:B,\Delta)$ = $\begin{array}{l} \mathsf{Zero}^c([]:0,\Delta) \\ \mathsf{Zero}^c([\mathsf{inl}\ p_1 \mid \mathsf{inr}\ p_2]:A+B,\Delta) \end{array}$ = ||Zero $^{c}(p_{1}:A,\Delta) |$ Zero $^{c}(p_{2}:B,\Delta)]$ = $\mathsf{Ex}(\Delta; \top : A; \Delta'; r)$ = $\mathsf{Ex}(\Delta; p_2 : A; \Delta', p_1 : A; \mathsf{Ex}(\Delta; p_1 : A; p_2 : A, \Delta'; r))$ $\mathsf{Ex}(\Delta; p_1 \wedge p_2 : A; \Delta'; r)$ = $\mathsf{Ex}(\Delta; \perp : A; \Delta'; r)$ $\mathsf{Zero}^{\perp}(\Delta, \Delta')$ = $\mathsf{Ex}(\Delta; p_1 \lor p_2 : A; \Delta'; r)$ $\mathsf{Join}^{\vee}(\Delta, \Delta'; \mathsf{Ex}(\Delta; p_1 : A; \Delta'; \mathsf{Out}_1^{\vee}(\Delta; r)); \mathsf{Ex}(\Delta; p_1 : A; \Delta'; \mathsf{Out}_2^{\vee}(\Delta; r)))$ = $\mathsf{Ex}(\Delta; x : A; \Delta'; r)$ = r $\mathsf{Ex}(\Delta; \langle \rangle : 1; \Delta'; r)$ = r $\mathsf{Ex}(\Delta; \langle p_1, p_2 \rangle : A \times B; \Delta'; r)$ $\mathsf{Ex}(\Delta; p_2 : B; \Delta', p_1 : A; \mathsf{Ex}(\Delta; p_1 : A; p_2 : B, \Delta'; r))$ = $\mathsf{Zero}^{[]}(\Delta, \Delta')$ $\mathsf{Ex}(\Delta; []: 0; \Delta'; r)$ = $= \operatorname{let} r_1 = \mathsf{Ex}(\Delta; \ p_1 : A; \ \Delta'; \ \mathsf{Out}_1^{[\cdot] \cdot]}(\Delta; \ r))$ $\mathsf{Ex}(\Delta; [\mathsf{inl} \ p_1 \mid \mathsf{inr} \ p_2] : A + B; \ \Delta'; \ r)$ let $r_2 = \mathsf{Ex}(\Delta; p_1 : A; \Delta'; \mathsf{Out}_2^{\overline{[\cdot]} \cdot]}(\Delta; r))$ $\mathsf{Join}^{[\cdot|\cdot]}(\Delta,\Delta'; r_1; r_2)$

Figure 9. Deep Operators

$$\frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PI} \ u \rightsquigarrow u'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PI} \ \lambda p'. u \rightsquigarrow \lambda p'. u'}} \qquad \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PI} \ r \rightsquigarrow r'}}$$

$$\frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ Out_{1}^{[\cdot]}(\Delta; r) \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}}{\frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}} \qquad \frac{\Delta}{\langle\langle v/p \rangle\rangle_{A}^{PE} \ r \rightsquigarrow r'}}$$

Figure 10. Deep Substitution

And_A(Δ ; \top ; c; r) = And_A(Δ ; c; \top ; r) = (c,r)And_A(Δ ; $x \wedge c_1$; c_2 ; r) = $\begin{array}{l} \operatorname{let}\left(c';r'\right) = \operatorname{And}_{A}(\Delta; \ c_{1}; \ c_{2}; \ r) \\ \operatorname{if} c' = y \wedge c'' \ \operatorname{then} \left(c'; [x/y]_{A}^{\operatorname{PE}} \ r\right) \ \operatorname{else}\left(c'; r\right) \end{array}$ And_A(Δ ; c_1 ; $x \wedge c_2$; r) = And₁(Δ ; $\langle \rangle$; x; r) = And₁(Δ ; x; $\langle \rangle$; r) = (x;r)And₀(Δ ; []; x; r) = $([]; \mathsf{Zero}^{[]}(\Delta))$ And₀(Δ ; x; []; r) = And_{$A \times B$}(Δ ; x; $\langle c_1, c_2 \rangle$; r) = $(x \land \langle c_1, c_2 \rangle; r)$ And_{$A \times B$}(Δ ; $\langle c_1, c_2 \rangle$; x; r) = And_{A+B}(Δ ; x; [inl c_1 | inr c_2]; r) = $\begin{array}{l} (x \wedge [\mathsf{inl} \ c_1 \ | \ \mathsf{inr} \ c_2]; r) \\ (x; [x/y]_A^{\mathsf{PE}} \ r) \end{array}$ And_{A+B}(Δ ; [inl c_1 | inr c_2]; x; r) = And_A(Δ ; x; y; r) = $\operatorname{And}_1(\Delta; \langle \rangle; \langle \rangle; r)$ = $(\langle \rangle; r)$ $([];\mathsf{Zero}^{[]}(\Delta))$ And₀(Δ ; []; []; r) = $\begin{array}{l} (\amalg, 2cio \ (\Delta)) \\ \text{let} (c_1''; r') &= \mathsf{And}_A(\Delta; c_1; c_1'; \mathsf{Ex}(\Delta, c_1 : A; c_2 : B; c_1' : A; r)) \\ \text{let} (c_2''; r'') &= \mathsf{And}_B(\Delta, c_1'' : A; c_2; c_2'; r') \\ (\langle c_1'', c_2'' \rangle; r'') \\ \end{array}$ And_{$A \times B$}(Δ ; $\langle c_1, c_2 \rangle$; $\langle c'_1, c'_2 \rangle$; r) = $\begin{array}{l} \operatorname{let} (c_1'';r_1'') = \operatorname{And}_A(\Delta; \ c_1; \ c_1'; \ \operatorname{Out}_1^{[\cdot|\cdot]}(\Delta, c_1:A; \ \operatorname{Out}_1^{[\cdot|\cdot]}(\Delta; \ r))) \\ \operatorname{let} (c_2'';r_2'') = \operatorname{And}_A(\Delta; \ c_2; \ c_2'; \ \operatorname{Out}_2^{[\cdot|\cdot]}(\Delta, c_1:A; \ \operatorname{Out}_2^{[\cdot|\cdot]}(\Delta; \ r))) \\ ([\operatorname{inl} \ c_1'' \ | \ \operatorname{inr} \ c_2'']; \ \operatorname{Join}^{[\cdot|\cdot]}(\Delta; \ r_1''; \ r_2'')) \end{array}$ And_{A+B}(Δ ; [inl c_1 | inr c_2]; [inl c'_1 | inr c'_2]; r) =

Figure 13. Conjunction Simplification

MergeTop(x; y; r) $= (y; [y/x]_A^{\mathsf{PE}} r)$ $MergeTop(x; \top; r)$ = (x;r) $(y \wedge c; [y/x]_A^{\mathsf{PE}} r)$ $\mathsf{MergeTop}(x; y \land c; r)$ = $\mathsf{MergeTop}(x;\langle\rangle;r)$ (x;r)= MergeTop(x; []; r)= ([];r) $MergeTop(x; \langle c_1, c_2 \rangle; r)$ = $(x \land \langle c_1, c_2 \rangle; r)$ MergeTop $(x; [inl c_1 | inr c_2]; r)$ $(x \wedge [\operatorname{inl} c_1 \mid \operatorname{inr} c_2]; r)$ = $\mathsf{MergeTop}_A(\top; \Delta; r)$ = $\mathsf{MergeTop}_A(x;\Delta;r)$ = $\mathsf{MergeTop}_A(x \wedge c; \Delta; r)$ $\mathsf{MergeTop}_A(c; \Delta; r)$ = $\mathsf{MergeTop}_1(\langle \rangle; \Delta; \Delta)r$ = $\mathsf{MergeTop}_0([]; \Delta; r)$ = $Zero[](\Delta)$ $\mathsf{MergeTop}_{A \times B}(\langle c_1, c_2 \rangle; \Delta; r)$ = $\mathsf{MergeTop}_B(c_2; \Delta, c_1 : A; \mathsf{MergeTop}_A(c_1; \Delta; r))$ $\mathsf{MergeTop}_{A+B}([\mathsf{inl}\ c_1 \mid \mathsf{inr}\ c_2]; \Delta; r)$ $\mathsf{Join}^{[\cdot|\cdot]}(\Delta; \mathsf{MergeTop}_A(c_1; \Delta; r); \mathsf{MergeTop}_A(c_2; \Delta; r))$ = $\operatorname{Merge}_{A}(\Delta_{1}; \top; r_{1}; \Delta_{2}; c_{2}; r_{2})$ = let $r'_1 = \text{MergeTop}_A(c_2; \Delta_1; r_1)$ $(c_2; r'_1; r_2)$ $= \operatorname{let} (c; r'_1; r'_2) = \operatorname{Merge}_A(\Delta_1; \top; r_1; \Delta_2; c_2; r_2)$ $\mathsf{Merge}_A(\Delta_1; x; r_1; \Delta_2; c_2; r_2)$ let $(c'; r''_1) = \mathsf{MergeTop}(x; c; r'_1)$ $(c'; r''_1; r'_2)$ $\begin{array}{l} (c';r_1';r_2') = \mathsf{Merge}_A(\Delta_1;c_1;r_1;\Delta_2;c_2;r_2) \\ \mathsf{let} \ (c';r_1') = \mathsf{MergeTop}(x;c;r_1') \\ (c';r_1'';r_2') \end{array}$ $\mathsf{Merge}_A(\Delta; x \wedge c_1; r_1; \Delta_2; c_2; r_2)$ = $\mathsf{Merge}_1(\Delta_1; \langle \rangle; r_1; \Delta_2; \langle \rangle; r_2)$ $(\langle \rangle; r_1; r_2)$ = $([]; \mathsf{Zero}^{[]}(\Delta_1); \mathsf{Zero}^{[]}(\Delta_2))$ $Merge_0(\Delta_1; []; r_1; \Delta_2; []; r_2)$ = $\begin{array}{l} \mathsf{let} \ (c_1'';r_1';r_2') = \mathsf{Merge}_A(\Delta_1;c_1;r_1;\Delta_2;c_1';r_2') \\ \mathsf{let} \ (c_2'';r_1'';r_2'') = \mathsf{Merge}_B(\Delta_1,c_1'':A;c_2;r_1';\Delta_2,c_1'':A;c_2';r_2') \\ \end{array}$ $\mathsf{Merge}_{A \times B}(\Delta_1; \langle c_1, c_2 \rangle; r_1; \Delta_2; \langle c_1', c_2' \rangle; r_2)$ $(\langle c_1'', c_2'' \rangle; r_1''; r_2'')$
$$\begin{split} & (c_1'';r_1';r_2') = \mathsf{Merge}_A(\Delta_1;c_1;\mathsf{Out}_1^{[\cdot]\cdot]}(\Delta_1;\ r_1);\Delta_2;c_1';\mathsf{Out}_1^{[\cdot]\cdot]}(\Delta_2;\ r_2')) \\ & \mathsf{let}\ (c_2'';r_1'';r_2'') = \mathsf{Merge}_B(\Delta_1;c_1;\mathsf{Out}_2^{[\cdot]\cdot]}(\Delta_1;\ r_1);\Delta_2;c_1';\mathsf{Out}_2^{[\cdot]\cdot]}(\Delta_2;\ r_2')) \end{split}$$
 $Merge_{A+B}(\Delta_1; [inl c_1 | inr c_2]; r_1; \Delta_2; [inl c'_1 | inr c'_2]; r_2)$ = $([\operatorname{inl} c_1'' \mid \operatorname{inr} c_2'']; \operatorname{Join}^{[\cdot|\cdot]}(\Delta; r_1'; r_1''); \operatorname{Join}^{[\cdot|\cdot]}(\Delta; r_2'; r_2''))$ $\mathsf{Merge}_{A}(\Delta_{2}; c_{2}; r_{2}; \Delta_{1}; c_{1}; r_{1})$ $Merge_{A}(\Delta_{1}; c_{1}; r_{1}; \Delta_{2}; c_{2}; r_{2})$ = (when the other cases don't match) $= (\cdot; r_1; r_2)$ $\mathsf{Merge}^*_{\cdot}(\Delta_1; \ \cdot; \ r_1; \ \Delta_2; \ \cdot; \ r_2)$ $= \operatorname{let} (c'; r'_1; r'_2) = \operatorname{Merge}_{A_1}(\Delta_1; c_1; r_1; A_2; \Delta_2; c_2) r_2$ $\mathsf{Merge}^*_{A_1 \overrightarrow{A}}(\Delta_1; c_1, \overrightarrow{c_1}; r_1; \Delta_2; c_2, \overrightarrow{c_2}; r_2)$ let $(\overrightarrow{c'}; r_1''; r_2'') = \mathsf{Merge}_{\overrightarrow{a}}^*(\Delta_1; \overrightarrow{c_1}; r_1'; \Delta_2; \overrightarrow{c_2}; r_2')$ $(c', \overrightarrow{c'}; r_1''; r_2'')$

Figure 14. Context Merging