Focusing on Binding and Computation

Daniel R. Licata* Noam Zeilberger Robert Harper*
Carnegie Mellon University
{drl,noam,rwh}@cs.cmu.edu

Abstract

Variable binding is a prevalent feature of the syntax and proof theory of many logical systems. In this paper, we define a programming language that provides intrinsic support for both representing and computing with binding. This language is extracted as the Curry-Howard interpretation of a focused sequent calculus with two *kinds of implication, of opposite polarity. The* representational arrow *extends systems of definitional reflection with a notion of scoped inference rules, which are used to represent binding. On the other hand, the usual* computational arrow *classifies recursive functions defined by pattern-matching. Unlike many previous approaches, both kinds of implication are connectives in a single logic, which serves as a rich logical framework capable of representing inference rules that mix binding and computation.*

1 Introduction

A *logical framework* provides a set of reusable abstractions that simplify the task of representing the syntax and semantics of *logical systems,* such as programming languages and proof theories. For example, the LF logical framework [17] permits facile representations of binding and scope (α-equivalence, capture-avoiding substitution) using the LF function space, a type which corresponds to logical implication. In this broad sense of the phrase, programming languages such as ML and Haskell are even more basic examples of logical frameworks, in that algebraic datatypes permit first-order representations of syntax and proofs. These languages' function spaces, which also correspond to logical implication, provide support not for representing binding, but for *computing* with syntax and proofs by pattern-matching. In contrast, LF requires a separate layer such as Twelf [29] for computation, which means that it is impossible to *embed* computations

*This research was sponsored in part by the National Science Foundation under grant number CCF-0702381. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

in LF representations—something easily achieved in ML or Haskell by defining a datatype with a function as a component. In a dependent programming language (e.g., Coq or Agda [10, 28]), embedding computations in data is especially useful, since it gives the full strength of iterated inductive definitions in the style of Martin-Löf [23]. Because the *same* logical connective, implication, is used to represent binding in LF, and to compute with binding in ordinary functional programming, it has proved difficult to integrate binding and computation in a single framework.

On the other hand, one way of distinguishing different aspects of the "same" connective is through the logical notion of *polarity* [15]. For example, linear logic exposes two conjunctions of opposite polarity (positive $\otimes$ and negative $\&$), and likewise two disjunctions (positive $\oplus$ and negative $\parr$). Operationally, polarity can be given an intuitive explanation in terms of pattern-matching [41]: values of positive polarity can be eliminated by pattern-matching against their constructors, whereas values of negative polarity can be *introduced* by pattern-matching against their *destructors.* This is why, for example, in ordinary functional programming, functions can be defined by pattern-matching, but it is impossible to pattern-match against a function (except with

a variable pattern)—implication is a negative connective.

Our work began with the observation that, although variable binding behaves in some ways like ordinary implication, it also seems to have positive polarity. For example, in Twelf, LF functions are analyzed by matching against higher-order patterns. Following this intuition, we define a logic that includes both a positive form of implication—used to represent binding—and ordinary negative implication—used to compute with binding. This logic builds on definitional reflection [16, 35], which supposes a database of rules used for both building proofs of propositional atoms and for deriving consequences of atoms by "reflection" (i.e., by inverting the rules). Through the Curry-Howard interpretation, the rule database corresponds to a database of datatype constructors, which can be used both to build datatype values and to define functions by pattern-matching. The key novelty of our positive implication is that it permits this database of rules to *vary.* Positive implication, written $R \Rightarrow A$, internalizes the act of hypothesizing a new rule: a proof of $R \Rightarrow A$ is a proof of A under assumption of the inference rule R. A value of type $R \Rightarrow A$ has the form $\lambda u.V$, where u is a *scoped datatype constructor.* Such a value is deconstructed by pattern-matching with a higher-order pattern. We call this positive connective $\Rightarrow$

the *representational arrow,* opposed to the negative connective $\rightarrow$, the ordinary *computational arrow.*

This approach to representing variable binding, which we call *definitional variation,* provides a more general theory of inference rules than LF, because rules can mix representational and computational functions. All rule systems representable in LF satisfy the structural properties of a hypothetical judgement (weakening, exchange, contraction, and substitution) because all LF rules are *pure* [4]—they place no constraints on the context in which they can be applied. In contrast, computational functions can be used to define impure rules: for example, if a rule system includes a rule with premise $P \rightarrow \bot$ asserting the refutability of P, it will not be possible to weaken a derivation using this rule to a context including a proof of P. However, this failure of the structural properties is not problematic in our framework: First, the representational arrow is eliminated by pattern matching, not by application (modus ponens), so the framework itself requires no commitment to the structural properties. Second, using a notion of subordination [39], we can give general conditions under which the structural properties hold , and provide operations such as weakening and substitution "for free" when these conditions are satisfied. In this sense our calculus maintains the practical ben-

efits of the LF approach, where the structural properties are provided by the framework, while providing a more general theory of inference rules.

Following Zeilberger [42], our computational arrow admits a form of open-endedness [20]: computational functions in our type theory are represented abstractly by meta-level functions from patterns to expressions. This open-endedness has several practical benefits: (1) We can implement structural properties such as weakening and substitution once as a datatype-generic program at the meta-level, reusing one implementation for a large class of rule systems. (2) We can realize meta-functions as programs in existing proof assistants, which permits us to reuse their pattern coverage checkers. (3) We can use our type theory as an interface for combining functions written in different proof assistants, using different implementations of binding, in a single program.

The technical contributions of this paper are as follows: In Section 2, we present a focused sequent calculus with both implications $\rightarrow$ and $\Rightarrow$, as well as a suite of other connectives. We define the identity and cut-elimination procedures, and prove they are total under assumptions about the form of the rule database. We discuss some counterintuitive logical properties of $\Rightarrow$, as well as a dual representational

conjunction. In Section 3, we give a proof term assignment to the sequent calculus, yielding a functional programming language with an operational semantics given by cut elimination. In Section 4, we show that our framework extends simply-typed LF and discuss datatype-generic implementations of the structural properties. In Section 5, we illustrate programming in our type theory with an example that mixes binding and computation; more examples are available in our companion technical report [22].

2 Sequent Calculus

When describing the sequent calculus in this section, we foreshadow the proof-term assignment given in Section 3, freely interchanging logical and type-theoretic terminology ("proposition" and "type", "implication" and "function space", "logic" and "type theory", etc.). The logic we construct is *polarized,* meaning that we maintain a syntactic separation between positive and negative propositions, and its proofs are *focalized* in the sense of Andreoli [3]. Following Zeilberger [41], the focused sequent calculus is defined in two stages. First, the polarized connectives are defined by axiomatizing the structure of *patterns.* Positive connectives are defined by *constructor patterns,* and nega-

tive connectives by *destructor patterns.* Second, there is a general focusing framework that is independent of the particular connectives of the logic.

For the sake of presentation, we begin by defining a focused sequent calculus for polarized intuitionistic logic, including the simple structure of patterns and the general focusing rules—this sequent calculus is a variation of the one given for polarized classical logic in [41]. We then extend the structure of patterns to describe the more expressive logic of definitional variation. Next, we prove the identity and cut theorems for this logic, and consider some interesting properties of the representational connectives.

2.1 Simple contexts and patterns

We write X^+, Y^+, Z^+ and X^-, Y^-, Z^- to stand for positive and negative propositional variables (atomic propositions), and A^+, B^+, C^+ and A^-, B^-, C^- to stand for arbitrary positive and negative formulas. We use α to range over *assumptions* X^+ or C^-, and dually γ to range over *conclusions* X^- or C^+. A *linear context* Δ is a list of assumptions.

The positive connectives are defined through the judgement $\Delta \Vdash C^+$, which corresponds to applying only linear right-rules to show C^+ from Δ. For example, the rules for

atoms, conjunction, and disjunction are as follows:

$$\frac{}{X^+ \Vdash X^+} \quad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash B^+}{\Delta_1, \Delta_2 \Vdash A^+ \otimes B^+} \quad \frac{\Delta \Vdash A^+}{\Delta \Vdash A^+ \oplus B^+} \quad \frac{\Delta \Vdash B^+}{\Delta \Vdash A^+ \oplus B^+}$$

Foreshadowing the Curry-Howard interpretation, we refer to derivations of this judgement as *constructor patterns;* linearity captures the restriction familiar from functional programming that a pattern binds a variable just once.

Negative connectives are defined by $\Delta \Vdash C^- > \gamma$, which corresponds to using linear left-rules to decompose C^- into the conclusion γ. A proof term for this judgement is a *destructor pattern*, which gives the shape of an elimination context (continuation) for negative types:

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash B^- > \gamma}{\Delta_1, \Delta_2 \Vdash A^+ \to B^- > \gamma} \quad \frac{\Delta \Vdash A^- > \gamma}{\Delta \Vdash A^- \& B^- > \gamma} \quad \frac{\Delta \Vdash B^- > \gamma}{\Delta \Vdash A^- \& B^- > \gamma}$$

Observe that a destructor pattern for $A^+ \to B^-$ includes a constructor pattern for A^+, as well as a destructor pattern for B^-, matching the possible observations on a function type. We have adopted linear logic notation by writing $\otimes$ for positive and $\&$ for negative conjunction. In the present setting, both of these connectives encode ordinary intuitionistic conjunction with respect to provability, but they have dif-

ferent proof terms: positive conjunction is introduced by an eager pair whose components are values, and eliminated by pattern-matching against both components; negative conjunction is eliminated by projecting one of the components, and introduced by pattern-matching against either possible observation, i.e. by a lazy pair.

2.2 Focusing Judgements

In Figure 1, we present the focusing rules. In these rules, Γ stands for a sequence of linear contexts Δ, but Γ itself is treated in an unrestricted manner (i.e., variables are bound once in a pattern, but may be used any number of times within the pattern's scope).

The first two judgements concern the positive connectives. The judgement $\Gamma \vdash [C^+]$ defines right-focus on a positive formula, or *positive values*: a positive value is a constructor pattern under a substitution for its free variables. Focus judgements make choices: to prove C^+ in focus, it is necessary to choose a particular shape of value by giving a constructor pattern, and then satisfy the pattern's free variables. Values are eliminated with the left-inversion judgement $\Gamma \vdash \gamma_0 > \gamma$, which defines a *positive continuation* by case-analysis. Inversion steps respond to all possible

choices that the corresponding focus step could make: the rule for C^+ quantifies over all constructor patterns for that formula, producing a result in each case. By convention, we tacitly universally quantify over metavariables such as Δ that appear first in a judgement that is universally quantified, so in full the premise reads "for all Δ, if $\Delta \Vdash C^+$ then $\Gamma, \Delta \vdash \gamma$." The positive connectives are thus introduced by choosing a value (focus) and eliminated by continuations that are prepared to handle any such value (inversion). For atoms, the only case-analysis is the identity.

Assumption	α	::=	$X^+ \mid C^-$
Conclusion	γ	::=	$X^- \mid C^+$
Linear context	Δ	::=	$\cdot \mid \Delta, \alpha$
Unrestricted context	Γ	::=	$\cdot \mid \Gamma, \Delta$

$$\boxed{\Gamma \vdash [C^+]} \qquad \frac{\Delta \Vdash C^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash [C^+]}$$

$$\boxed{\Gamma \vdash \gamma_0 > \gamma} \qquad \frac{}{\Gamma \vdash X^- > X^-} \qquad \frac{\forall (\Delta \Vdash C^+) : \Gamma, \Delta \vdash \gamma}{\Gamma \vdash C^+ > \gamma}$$

$$\boxed{\Gamma \vdash [C^-] > \gamma} \qquad \frac{\Delta \Vdash C^- > \gamma_0 \quad \Gamma \vdash \Delta \quad \Gamma \vdash \gamma_0 > \gamma}{\Gamma \vdash [C^-] > \gamma}$$

$$\forall (\Delta \Vdash C^- > \gamma) : \Gamma, \Delta \vdash \gamma \qquad X^+ \in \Gamma$$

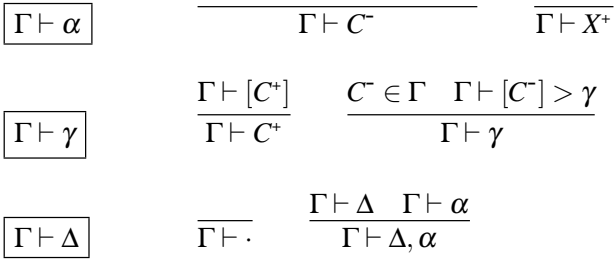


Figure 1. Focusing rules

The next two judgements concern the negative connectives, where the relationship between introduction/elimination and focus/inversion is reversed. A negative formula is *eliminated* by the left-focus judgement $\Gamma \vdash [C^-] > \gamma$, which chooses how to observe C^- by giving a *negative continuation*. A negative continuation consists of a destructor pattern, a substitution, and a case-analysis. The destructor pattern and substitution decompose a negative type C^- to some conclusion γ_0, for instance a positive type C^+. However, it may take further case-analysis of this positive type to reach the desired conclusion γ. Dually, negative types are *introduced* by inversion, which responds

to left-focus by giving sufficient evidence to support all possible observations. The right-inversion judgement $\Gamma \vdash \alpha$, where assumptions α are negative formula or positive atoms, specifies the structure of a *negative value*. A negative value for C^- must show that for all destructors of C^-, the conclusion is justified by the variables bound by the patterns in it.

The judgement $\Gamma \vdash \gamma$, defines a neutral sequent, or an *expression*: from a neutral sequent, one can either right-focus and return a value, or left-focus on an assumption in Γ and apply a negative continuation to it. Finally, a *substitution* $\Gamma \vdash \Delta$ provides a negative value for each hypothesis.

At this point, the reader may wish to work through some instances of these rules (using the above pattern rules) to see that they give the expected derived rules for the connectives:

Pos. formula	A^+	::=	$X^+ \mid \downarrow A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+$ $\mid P \mid R \Rightarrow B^+$
Rule	R	::=	$P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$
Neg. formula	A^-	::=	$X^- \mid \uparrow A^+ \mid A^+ \to B^- \mid \top \mid A^- \& B^-$ $\mid R \nleftarrow B^-$
Rule Context	Ψ	::=	$\cdot \mid \Psi, R$
Contextual form.	$C^\pm$	::=	$\langle \Psi \rangle A^\pm$

$\boxed{\Delta; \Psi \Vdash A^+}$

$$\frac{}{X^+ ; \Psi \Vdash X^+} \qquad \frac{}{\langle \Psi \rangle A^- ; \Psi \Vdash \downarrow A^-}$$

$$\frac{}{\cdot ; \Psi \Vdash 1} \qquad \frac{\Delta_1 ; \Psi \Vdash A^+ \quad \Delta_2 ; \Psi \Vdash B^+}{\Delta_1, \Delta_2 ; \Psi \Vdash A^+ \otimes B^+}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta ; \Psi \Vdash A^+}{\Delta ; \Psi \Vdash A^+ \oplus B^+} \qquad \frac{\Delta ; \Psi \Vdash B^+}{\Delta ; \Psi \Vdash A^+ \oplus B^+}$$

$$\frac{\Delta ; \Psi, R \Vdash B^+}{\Delta ; \Psi \Vdash R \Rightarrow B^+} \qquad \frac{P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+ \in \Psi \quad \Delta_1 ; \Psi \Vdash A_1^+ \quad \ldots \quad \Delta_n ; \Psi \Vdash A_n^+}{\Delta_1, \ldots, \Delta_n ; \Psi \Vdash P}$$

$$\boxed{\Delta ; \Psi \Vdash A^- > \gamma}$$

$$\frac{}{\cdot ; \Psi \Vdash X^- > X^-} \qquad \frac{}{\cdot ; \Psi \Vdash \uparrow A^+ > \langle \Psi \rangle A^+}$$

$$\text{(no rule for } \top\text{)} \qquad \frac{\Delta ; \Psi \Vdash A^- > \gamma}{\Delta ; \Psi \Vdash A^- \& B^- > \gamma} \qquad \frac{\Delta ; \Psi \Vdash B^- > \gamma}{\Delta ; \Psi \Vdash A^- \& B^- > \gamma}$$

$$\frac{\Delta_1 ; \Psi \Vdash A^+ \quad \Delta_2 ; \Psi \Vdash B^- > \gamma}{\Delta_1, \Delta_2 ; \Psi \Vdash A^+ \rightarrow B^- > \gamma} \qquad \frac{\Delta ; \Psi, R \Vdash B^- > \gamma}{\Delta ; \Psi \Vdash R \curlywedge B^- > \gamma}$$

We write $\Delta \Vdash \langle \Psi \rangle A^+$ for $\Delta ; \Psi \Vdash A^+$
and $\Delta \Vdash \langle \Psi \rangle A^- > \gamma$ for $\Delta ; \Psi \Vdash A^- > \gamma$.

Figure 2. Patterns

$$\frac{\Gamma \vdash X^- \quad \Gamma \vdash Y^- \quad \Gamma \vdash Z^-}{\Gamma \vdash (X^- \& Y^-) \& Z^-} \qquad \frac{\Gamma, X^+ \vdash Z^- \quad \Gamma, Y^+ \vdash Z^-}{\Gamma \vdash (X^+ \oplus Y^+) \to Z^-}$$

2.3 Patterns for Definitional Variation

In Section 2.1, we gave a fixed set of rules for constructing simple patterns. We now describe patterns for definitional variation by including an open-ended database of rules. A *rule* R takes the form $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$, where $A_1^+, \ldots, A_n^+$ are positive formulas and P is a *defined atom*. Rules are collected in a *rule context* Ψ, which is now carried through the pattern-typing judgments ($\Delta ; \Psi \Vdash A^+$ and $\Delta ; \Psi \Vdash A^- > \gamma$).

A rule $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+ \in \Psi$ can be applied to produce a constructor pattern for P:

$$\frac{\Delta_1 ; \Psi \Vdash A_1^+ \quad \ldots \quad \Delta_n ; \Psi \Vdash A_n^+}{\Delta_1, \ldots, \Delta_n ; \Psi \Vdash P}$$

Note that rules can be applied an arbitrary number of times while constructing a pattern. Now, consider the pattern-typing rules for the new connectives of definitional variation, *representational implication* and *representational con-*

junction:

$$\frac{\Delta\,;\,\Psi, R \Vdash B^+}{\Delta\,;\,\Psi \Vdash R \Rightarrow B^+} \qquad \frac{\Delta;\Psi, R \Vdash B^- > \gamma}{\Delta;\Psi \Vdash R \curlywedge B^- > \gamma}$$

Both connectives expand the rule context, introducing a scoped constructor of type R. The rule for $R \Rightarrow B^+$ builds a constructor pattern for B^+ under assumption of R and essentially (if we ignore structural punctuation) looks like an implication right-rule, while the rule for $R \curlywedge B^-$ builds a destructor pattern for B^- and looks like a conjunction left-rule. However, as we will see in Section 2.5, these connectives behave quite differently from ordinary implication and conjunction, in part due to their non-standard polarity.

Most of the remaining rules (see Figure 2) for the connectives of polarized logic are unremarkable, since they simply carry the rule context through unchanged. The "shift" connectives $\uparrow$ and $\downarrow$ deserve explanation, though. Following Girard [14], these mark the boundary between positive and negative polarity, and correspondingly they mark the point where pattern-matching must end [41]. Because the rule context can change during the course of pattern-matching, it is necessary to associate assumptions and conclusions with a specific rule context. We indicate this with *contextual formulas* $\langle\Psi\rangle A^+$ and $\langle\Psi\rangle A^-$, so that the

rules for the shift connectives are:

$$\frac{}{\langle\Psi\rangle A^- ; \Psi \Vdash \downarrow A^-} \qquad \frac{}{\cdot ; \Psi \Vdash \uparrow A^+ > \langle\Psi\rangle A^+}$$

In spite of this richer notion of patterns, the generic focusing rules of Figure 1 remain unchanged if we adopt some notational sleight-of-hand: we now take C^+ and C^- to range over contextual formulas, and write $\Delta \Vdash \langle\Psi\rangle A^+$ as notation for $\Delta ; \Psi \Vdash A^+$, and $\Delta \Vdash \langle\Psi\rangle A^- > \gamma$ for $\Delta ; \Psi \Vdash A^- > \gamma$.

Example Consider the syntax of the untyped λ-calculus: $e \quad ::= \quad x \mid \lambda x.e \mid e_1\, e_2$ This syntax is represented in our type theory by the following definition signature Ψ_λ:

$$\mathsf{lam} : \mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp}) ; \mathsf{app} : \mathsf{exp} \Leftarrow \mathsf{exp} \Leftarrow \mathsf{exp}$$

For clarity, we name the rules in the rule context here, foreshadowing the presentation with proof terms in Section 3. The λ-calculus terms with free variables $x_1, \ldots, x_n$ are isomorphic to derivations of the constructor pattern judgement $\cdot ; \Psi_\lambda, x_1 : \mathsf{exp}, \ldots, x_n : \mathsf{exp} \Vdash \mathsf{exp}$. The fact that the rules defining exp may vary during a derivation is essential to this representation of the new variables bound in a term. The *computational* arrow then provides the means to induct over λ-terms: A negative value $\cdot \vdash \langle\Psi_\lambda\rangle\, \mathsf{exp} \rightarrow \uparrow \mathsf{exp}$

represents a function from λ-terms in the empty context to λ-terms in the empty context. Such a term is defined by an ω-rule which gives one case for each λ-term: whereas the traditional definitional reflection rule [16, 35] unrolls a definition only a single step, our inversion rules unroll a definition until they reach a polarity shift.

2.4 Identity and Cut

In addition to inductive types like exp, the context Ψ can be used to define arbitrary recursive types. For example, consider an atom D defined by one constant

$$\mathsf{d} : \mathsf{D} \Leftarrow \downarrow(\mathsf{D} \to \uparrow\mathsf{D})$$

D is essentially the recursive type $\mu X.X \to X$, which can be used to write non-terminating programs.

Because the rule context permits the definition of general recursive types, it should not be surprising that the identity and cut principles are not admissible in general. Through the Curry-Howard interpretation, however, we can still make sense of the identity and cut principles as corresponding, respectively, to the possibly infinite *processes* of η-expansion and β-reduction. We now state these prin-

ciples, "prove" admissibility of cut with an operationally sound but possibly non-terminating procedure (see our technical report [22] for the analogous identity procedure), and then discuss criteria under which this proof is well-founded.

Principle 1 (Identity)**.**

1. *(neg. identity) If* $C^- \in \Gamma$ *then* $\Gamma \vdash C^-$.
2. *(pos. identity)* $\Gamma \vdash C^+ > C^+$
3. *(identity substitution) If* $\Delta \subseteq \Gamma$ *then* $\Gamma \vdash \Delta$.

Principle 2 (Cut)**.**

1. *(neg. reduction) If* $\Gamma \vdash C^-$ *and* $\Gamma \vdash [C^-] > \gamma$ *then* $\Gamma \vdash \gamma$.
2. *(pos. reduction) If* $\Gamma \vdash [C^+]$ *and* $\Gamma \vdash C^+ > \gamma$ *then* $\Gamma \vdash \gamma$.
3. *(composition)*
   (a) *If* $\Gamma \vdash \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash \gamma$.
   (b) *If* $\Gamma \vdash [C^-] > \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash [C^-] > \gamma$.
   (c) *If* $\Gamma \vdash \gamma_1 > \gamma_0$ *and* $\Gamma \vdash \gamma_0 > \gamma$ *then* $\Gamma \vdash \gamma_1 > \gamma$.
4. *(substitution) For all six focusing judgements* J*, if* $\Gamma \vdash \Delta$ *and* $\Gamma, \Delta \vdash J$ *then* $\Gamma \vdash J$.

Procedure. Consider the first cut principle. The two derivations must take the following form:

$$\frac{\forall(\Delta \Vdash C^- > \gamma_0): \quad \Gamma, \Delta \vdash \gamma_0}{\Gamma \vdash C^-} \qquad \frac{\Delta \Vdash C^- > \gamma_0 \quad \Gamma \vdash \Delta \quad \Gamma \vdash \gamma_0 > \gamma}{\Gamma \vdash [C^-] > \gamma}$$

By plugging $\Delta \Vdash C^- > \gamma_0$ from the right derivation into the higher-order premise of the left derivation, we obtain $\Gamma, \Delta \vdash \gamma_0$. Then $\Gamma \vdash \gamma_0$ by substitution with $\Gamma \vdash \Delta$, whence $\Gamma \vdash \gamma$ by composition with $\Gamma \vdash \gamma_0 > \gamma$. The case of positive reduction is analogous (but appeals only to substitution).

In all cases of composition, if $\gamma_0 = X^-$ then the statement is trivial. Otherwise, we examine the last rule of the left derivation. For the first composition principle, there are two cases: either the sequent was derived by right-focusing on the conclusion $\gamma_0 = C^+$, or else by left-focusing on some hypothesis $C^- \in \Gamma$. In the former case, we immediately appeal to positive reduction. In the latter case, we apply the second composition principle, which in turn reduces to the third, which then reduces back to the first.

Likewise, to show substitution we examine the rule concluding $\Gamma, \Delta \vdash J$. Dually to the composition principle, the only interesting case is when the sequent was derived by

left-focusing on $C^- \in \Delta$, wherein we immediately apply a negative reduction.

□

Observe that we have made no mention of particular connectives or rule contexts, instead reasoning uniformly about focusing derivations. As we alluded to above, however, in general this procedure is not terminating. Here we state sufficient conditions for termination. They are stated in terms of a *strict subformula ordering,* a more abstract version of the usual structural subformula ordering.

Definition 1 (Strict subformula ordering)**.** *We define an ordering $C_1^\pm \sqsupset C_2^\pm$ between contextual formulas as the least transitive relation closed under the following properties:*

- *If $\Delta \Vdash C_1^- > \gamma$ and $C_2^- \in \Delta$ then $C_1^- \sqsupset C_2^-$*
- *If $\Delta \Vdash C_1^- > \gamma$ and $C_2^+ = \gamma$ then $C_1^- \sqsupset C_2^+$*
- *If $\Delta \Vdash C_1^+$ and $C_2^- \in \Delta$ then $C_1^+ \sqsupset C_2^-$*

For any contextual formula $C^\pm$, we define $\sqsupset_C$ to be the restriction of $\sqsupset$ to formulas below $C^\pm$.

The strict subformula ordering does not mention atoms X^+ or X^-, since they only play a trivial role in identity and cut.

Definition 2 (Well-founded formulas)**.** *We say that a contextual formula $C^{\pm}$ is well-founded if $\sqsupset_C$ is well-founded.*

Proposition 1. *Positive and negative identity are admissible on well-founded formulas.*

Proposition 2. *Positive and negative reduction are admissible on well-founded formulas.*

Proof. By inspection of the above procedure. Positive and negative reduction are proved by mutual induction using the order $\sqsupset_C$, with a side induction on the left derivation to show composition, and a side induction on the right derivation to show substitution. □

Definition 3 (Pure rules)**.** *A rule R is called* pure *if it contains no shifted negative formulas $\downarrow A^-$ as premises (or structural subformulas of premises). For example, $\mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp})$ is pure, but $\mathsf{D} \Leftarrow \downarrow(\mathsf{D} \rightarrow \uparrow D)$ is not.*

Lemma 1. *Suppose $\langle \Psi \rangle A^{\pm}$ contains only pure rules (i.e., in Ψ, or as structural subformulas of $A^{\pm}$). Then $\langle \Psi \rangle A^{\pm}$ is well-founded.*

Proof. By induction on the structure of $A^{\pm}$. Every pattern typing rule (recall Figure 2) examines only structural subformulas of $A^{\pm}$, except when $A^+ = P$. But any P defined

by pure rules $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$ in fact has *no* strict subformulas, since the Δ_i such that $\Delta_i ; \Psi \Vdash A_i^+$ can contain only atomic formulas X^+. □

The restriction to pure rules precludes premises involving the computational arrow. However, as we show below, it includes all inference rules definable in the LF logical framework, generalizing Schroeder-Heister's [35] proof of cut-elimination for the fragment of definitional reflection with $\rightarrow$-free rules (since pure rules do *not* exclude $\Rightarrow$'s). Moreover, as we explained, the identity and cut principles are always operationally meaningful, even in the presence of arbitrary recursive types. Technically, we could adopt a coinductive reading of the focusing rules (cf. [14]), in which case identity is always productive, and cut-elimination is a partial operation that attempts to build a cut-free proof bottom-up. We conjecture that cut-elimination is total given a positivity restriction for rules.

2.5 Shock therapy

In §6.2 of "Locus Solum", Girard [14] considers several "shocking equalities"—counterintuitive properties of the universal and existential quantifiers that emerge when they are given non-standard polarities. For example, posi-

tive $\forall$ commutes under $\oplus$, while negative $\exists$ commutes over $\&$. In our setting, $\Rightarrow$ behaves almost like a positive universal quantifier, and $\curlywedge$ almost like a negative existential.[1] And indeed, we can reproduce analogues of these commutations.

Definition 4. *For two positive contextual formulas C_1^+ and C_2^+, we say that $C_1^+ \lesssim C_2^+$ if $\cdot \vdash C_1^+ > C_2^+$. For negative C_1^- and C_2^-, we say $C_1^- \lesssim C_2^-$ if $C_1^- \vdash C_2^-$. We write $C_1^\pm \approx C_2^\pm$ when both $C_1^\pm \lesssim C_2^\pm$ and $C_2^\pm \lesssim C_1^\pm$. These relations are extended to (non-contextual) polarized formulas if they hold under all rule contexts.*

Proposition 3 ("Shocking" equalities)**.**

1. $R \Rightarrow (A^+ \oplus B^+) \approx (R \Rightarrow A^+) \oplus (R \Rightarrow B^+)$
2. $(R \curlywedge A^-) \& (R \curlywedge B^-) \approx R \curlywedge (A^- \& B^-)$

[1] These would become real quantifiers in an extension to dependent types.

Rule Context	Ψ	::=	$\cdot \mid \Psi, u{:}R$
Con. Pattern	p	::=	$x \mid () \mid (p_1, p_2) \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p$
			$\mid u\, p_1 \ldots p_n \mid \lambda\, u.p$
Dest. Pattern	n	::=	$\varepsilon \mid p; n \mid \mathsf{fst}; n \mid \mathsf{snd}; n \mid \mathsf{unpack}; u.n$
Context. Pat.	c	::=	$\overline{\Psi}.p$
	d	::=	$\overline{\Psi}.n$
Pos. Value	v^+	::=	$c\,[\sigma]$

Pos. Cont.	k^+	::=	$\varepsilon \mid \mathsf{val}^+(\phi^+) \mid \boxed{\ulcorner\ \bar{\varepsilon}\ \urcorner} \mid \boxed{k_1^+ ; k_2^+}$
			where $\phi^+ ::= \{c \mapsto e \mid \cdots\}$
Neg. Cont.	k^-	::=	$d[\sigma]; k^+ \mid \boxed{k^- ; k^+}$
Neg. Value	v^-	::=	$x \mid \mathsf{val}^-(\phi^-) \mid \boxed{\ulcorner\ \bar{x}\ \urcorner} \mid \boxed{\mathsf{fix}(x.v^-)}$
			where $\phi^- ::= \{d \mapsto e \mid \cdots\}$
Expression	e	::=	$v^+ \mid x \bullet k^- \mid \boxed{v^- \bullet k^- \mid v^+ \bullet k^+ \mid e ; k^+}$
Substitution	σ	::=	$\cdot \mid \sigma, v^-/x \mid \boxed{\ulcorner\ \mathsf{id}\ \urcorner} \mid \boxed{\sigma_1, \sigma_2}$

Figure 3. Proof Terms

Proof. Immediate—indeed, in each case, both sides have an isomorphic set of patterns. □

Why are these equalities shocking? Well, if we ignore polarity and treat all the connectives as ordinary implication, disjunction, and conjunction, then (2) is reasonable but (1) is only valid in classical logic. And if we interpret $\Rightarrow$ and $\curlywedge$ as $\forall$ and $\exists$, then both equations are shockingly anticlassical:

1. $\forall x.(A \oplus B) \approx (\forall x.A) \oplus (\forall x.B)$
2. $(\exists x.A) \& (\exists x.B) \approx \exists x.(A \& B)$

On the other hand, from a computational perspective, these

equalities are quite familiar. For example, (1) says that a value of type $A \oplus B$ with a free variable is either the left injection of an A with a free variable or the right injection of a B with a free variable.

We can state another pair of surprising equivalences *between* the connectives $\Rightarrow$ and $\curlywedge$ under polarity shifts:

Proposition 4 (Some/any)**.**

1. $\downarrow(R \curlywedge A^-) \approx R \Rightarrow \downarrow A^-$
2. $\uparrow(R \Rightarrow A^+) \approx R \curlywedge \uparrow A^+$

Again, this coincidence under shifts is not *too* surprising, since it recalls the some/any quantifier $И x.A$ of nominal logic [31], as well as the self-dual ∇ connective of Miller and Tiu [25]. $И x.A$ can be interpreted as asserting either that A holds for some fresh name, or for *all* fresh names—with both interpretations being equivalent.

3 Proof Terms

In Figure 3, we present a proof term assignment to the focused sequent calculus described above, with one proof

$\boxed{e \hookrightarrow e'}$

$$\frac{\phi^+(c)\,\mathsf{defined}}{c\,[\sigma] \bullet \mathsf{val}^+(\phi^+) \hookrightarrow \phi^+(c)\,[\sigma]}\ \mathrm{pr}$$

$$\frac{}{v^+ \bullet (k_1^+ ; k_2^+) \hookrightarrow (v^+ \bullet k_1^+) ; k_2^+}\ \mathrm{k^+k^+}$$

$$\frac{}{v^+ \bullet \varepsilon \hookrightarrow v^+}\ \mathrm{idk^+}$$

- -

$$\frac{\phi^-(d)\,\mathsf{defined}}{\mathsf{val}^-(\phi^-) \bullet (d[\sigma]; k^+) \hookrightarrow (\phi^-(d)\,[\sigma]) ; k^+}\ \mathrm{nr}$$

$$\frac{}{v^- \bullet (k^- ; k^+) \hookrightarrow (v^- \bullet k^-) ; k^+}\ \mathrm{k^+k^-}$$

$$\frac{}{\mathsf{fix}(x.v^-) \bullet k^- \hookrightarrow v^-\,[\mathsf{fix}(x.v^-)/x] \bullet k^-}\ \mathrm{fix}$$

- -

$$\frac{e \hookrightarrow e'}{e ; k^+ \hookrightarrow e' ; k^+}\ \mathrm{k^+ee} \qquad \frac{}{v^+ ; k^+ \hookrightarrow v^+ \bullet k^+}\ \mathrm{k^+ev}$$

Figure 4. Operational Semantics

term for each rule in the calculus. Additionally, we internalize the cut and identity principles: $v^- \bullet k^-$ and $v^+ \bullet k^+$ witness reduction; $e\,;k^+$ and $k^-\,;k^+$ and $k_1^+\,;k_2^+$ witness composition; and x and ε and id witness identity. For programming convenience (see Section 5), we also internalize an admissible substitution concatenation principle (σ_1, σ_2), and a general recursion operator $\mathsf{fix}(x.v^-)$. To make the examples below more concise, we tacitly parametrize all judgements by a fixed initial definition context Σ, which acts as a prefix on each contextual formula in the judgement forms (i.e., $\langle\Psi\rangle A$ acts as $\langle\Sigma, \Psi\rangle A$ did without the signature). The full typing rules are presented in Figures 6 and 7 at the end of this article.

α-equivalence The pattern for a contextual type $\langle\Psi\rangle A^+$ is a contextual pattern $\overline{\Psi}.p$, where $\overline{\Psi}$ notates the bare variables (no rule annotations) from Ψ. Contextual patterns for $\langle\Psi\rangle A^+$ are modal [27]—all of the rule variables free in the pattern must be explicitly bound by Ψ—and are typed by the judgement $\Delta \Vdash \overline{\Psi}.p :: \langle\Psi\rangle A^+$. This judgement is defined by passing to the judgement $\Delta\,;\Psi \Vdash p :: A^+$, in which the variables in Ψ are free in p. Negative patterns $\overline{\Psi}.n$ are typed similarly:

$$\frac{\Delta;\Psi \Vdash p :: A^+}{\Delta \Vdash \overline{\Psi}.p :: \langle\Psi\rangle A^+} \qquad \frac{\Delta;\Psi \Vdash n :: A^- > \gamma}{\Delta \Vdash \overline{\Psi}.n :: \langle\Psi\rangle A^- > \gamma}$$

In the sequent calculus above, we treated the judgements $\Delta \Vdash \langle\Psi\rangle A^+$ and $\Delta;\Psi \Vdash A^+$ synonymously, but here this distinction clarifies the binding structure of our language. The proof terms $\overline{\Psi}.p$ and $\overline{\Psi}.n$, as well as the proof terms $\lambda\, u.p$ and $\mathsf{unpack}; u.n$ for $\Rightarrow$ and $\curlywedge$, are binding forms, and the standard notion of α-equivalence applies to them and to the context Ψ in the typing judgements for p and n. Other contexts Ψ' appear in contextual types in Δ and γ, but these are separate binding occurrences and can be renamed independently.[2] Similarly, in $\Delta \Vdash \overline{\Psi}.p :: \langle\Psi\rangle A^+$, the variables in $\overline{\Psi}$ and in Ψ are independent binding occurrences. Because patterns are modal, no rule variables are free in values, continuations, expressions, or substitutions. We tacitly quotient patterns by α-equivalence at the meta-level, so that meta-functions are defined on α-equivalence classes of patterns. This ensures that computational functions respect α-equivalence of represented languages.

Meta-functions. Our type theory is, by design, open-ended with respect to the meta-functions ϕ, mapping patterns to expressions, which are used to represent case-

analysis and induction. We have exploited this freedom by implementing simple embeddings of our language in Agda and Coq,[3] where meta-functions are realized as functions in Agda/Coq, and totality of meta-functions is established using the pattern coverage checkers of these existing tools. Both of these embeddings use de Bruijn indices to represent rule variables, but other implementations of our type theory are free to use different representations of variables, and program fragments written using different representations of binding can be combined.

Operational Semantics In Figure 4, we adapt the above cut-elimination procedure into a small-step operational semantics on closed expressions. We use an auxiliary meta-operation $e\,[\sigma]$ implementing capture-avoiding substitution, which is defined using the induction principle for the iterated inductive definition of our proof term syntax. Consequently, the operational semantics require that the the class of meta-functions ϕ is closed under definitions using this induction principle.

Theorem 1 (Type safety)**.**

Progress: *If* $\cdot \vdash e : \gamma$ *then* $e = v^+$ *or* $e \hookrightarrow e'$.

Preservation: *If* $\cdot \vdash e : \gamma$ *and* $e \hookrightarrow e'$ *then* $\cdot \vdash e' : \gamma$

4 Adequacy and Structural Properties

4.1 Embedding of Simply-Typed LF

The canonical forms of simply-typed LF (STLF) [40] are summarized in Figure 5. We show that the STLF terms exist as closed patterns, and therefore as values, in our type

[2] In our simply-typed setting, these contexts need not carry variables at all, but the variables would be necessary for dependency.

[3] Available from `http://www.cs.cmu.edu/~drl/`

Type	τ	::=	$P \mid \tau_1 \supset \tau_2$
Canonical Form	M	::=	$x\, M_1 \ldots M_n \mid \lambda x.M$
Signature	Σ	::=	$\cdot \mid \Sigma, x:\tau$
Context	Φ	::=	$\cdot \mid \Phi, x:\tau$

$$\frac{\Phi, x:\tau_1 \vdash_\Sigma M:\tau_2}{\Phi \vdash_\Sigma \lambda x.M : \tau_1 \supset \tau_2} \qquad \frac{x:\tau_1 \supset \ldots \supset \tau_n \supset P \in \Sigma \text{ or } \Phi \quad \Phi \vdash_\Sigma M_1:\tau_1 \quad \ldots \quad \Phi \vdash_\Sigma M_1:\tau_n}{\Phi \vdash_\Sigma x\, M_1 \ldots M_n : P}$$

Figure 5. Simply-typed LF

theory. This theorem permits us to inherit en masse the ade-

quacy of all systems that have been represented in STLF—e.g., the above signature Ψ_λ, which is the embedding of the usual LF encoding of this syntax.

Every STLF type τ can be parsed both as an inference rule $\mathsf{r}(\tau)$ and as a positive formula $\mathsf{p}(\tau)$ (for convenience, we identify STLF base types with our defined atoms P):

$$\mathsf{r}(\tau_1 \supset \ldots \supset \tau_n \supset P) \quad = \quad P \Leftarrow \mathsf{p}(\tau_1) \Leftarrow \ldots \Leftarrow \mathsf{p}(\tau_n)$$

$$\begin{array}{rcl} \mathsf{p}(P) & = & P \\ \mathsf{p}(\tau_1 \supset \tau_2) & = & \mathsf{r}(\tau_1) \Rightarrow \mathsf{p}(\tau_2) \end{array}$$

The function $\mathsf{r}(\tau)$ can then be used to map STLF signatures Σ and contexts Φ to inference rule contexts Ψ.

Theorem 2 (Embedding of STLF)**.** *Let* $\mathsf{r}(\Sigma) = \Psi_\Sigma$ *and* $\mathsf{r}(\Phi) = \Psi_\Phi$ *and* $\mathsf{p}(\tau) = A^+$*. Then there is a bijection between canonical STLF terms* M *such that* $\Phi \vdash_\Sigma M : \tau$ *and patterns* p *such that* $\cdot ; \Psi_\Phi \Vdash p :: A^+$ *in signature* Ψ_Σ*.*

Proof. Map $\lambda x.M$ to $\lambda x.p$ and $x M_1 \ldots M_n$ to $x\, p_1 \ldots pn$. □

To check that LF substitution is faithfully modelled in our calculus, we can recast the usual hereditary substitution algorithm for LF [40] as a meta-operation on closed pat-

terns. However, it is also possible to prove a much more general substitution principle, which covers many uses of iterated inductive definitions.

4.2 Structural Properties

As discussed above, the rule context Ψ does not in general satisfy the structural properties of a hypothetical judgement, because computational functions can be used to define impure rules. However, we can establish the structural properties generically under sufficient conditions that computational functions cannot interfere. To state these conditions, we use a notion of subordination [39], which tracks when values of one type are relevant to values of another. First, we define a judgement $P \not\preceq A^{\pm} \in \Psi$ ("P is insubordinate to $A^{\pm}$"), which means that no rule concluding P can be used by a value of type $A^{\pm}$. Next, we define a judgement $P \succeq A^{\pm} \in \Psi$ ("P is uncircumscribed by $A^{\pm}$"), which means that P is insubordinate to the domain of any computational arrow in $A^{\pm}$. We say that a rule $P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$ is insubordinate to/uncircumscribed by $A^{\pm}$ iff P is. Finally, we define a judgement $\mathsf{bindsof}\, A^{\pm} \succeq B^{\pm} \in \Psi$, which means that all rules that may be bound by a value of type $A^{\pm}$ are uncircumscribed by $B^{\pm}$. We refer the interested reader to our

companion Agda code for the formal definitions of these judgements.

Let $R = P \Leftarrow A_1^+ \cdots \Leftarrow A_n^+$. Using our Agda implementation, we have given negative values of the following types:

- *strengthen*: $\langle \Psi \rangle\, ((R \Rightarrow A^\pm) \to A^\pm)$ if $P \not\preceq A^\pm \in \Psi, u{:}R$.
- *weaken*: $\langle \Psi \rangle\, (A^\pm \to (R \Rightarrow A^\pm))$ if $P \succeq A^\pm \in \Psi, u{:}R$.
- *apply*: $\langle \Psi \rangle\, (((P \Rightarrow A^\pm) \otimes P) \to A^\pm)$ if $P \succeq A^\pm \in \Psi, u{:}P$ and $\mathsf{bindsof}\, A^\pm \succeq P \in (\Psi, u{:}P)$.

The function *strengthen* removes an insubordinate rule from the context. The function *weaken* adds an uncircumscribed rule to the context. The value *apply* substitutes a value for a base type; the subordination conditions are necessary for strengthening the arguments to computational functions in $A^\pm$ ($P \succeq A^\pm \in \Psi, u{:}P$) and weakening the proof of P as the substitution operation passes under binders in $A^\pm$ ($\mathsf{bindsof}\, A^\pm \succeq P \in (\Psi, u{:}P)$). These functions are defined by the same recursion on types as the proof of the identity theorem in Section 2, and they are total/productive in the same circumstances as identity. At present, we have only implemented substitution for base types generically, though we conjecture a generalization to all higher-order rules in the embedding of LF.

To illustrate these structural properties, consider a signature Ψ with constants $\mathsf{lam} : \mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp})$ and $\mathsf{omega} : \mathsf{exp} \Leftarrow \downarrow(\mathsf{nat} \to \uparrow\mathsf{exp})$, which might arise in representing a proof theory for natural numbers with an ω-rule. In this signature, exp is insubordinate to nat (expressions cannot be used to build natural numbers), but exp is not insubordinate to exp (expressions can be used to build expressions). Thus, *strengthen* permits strengthening away exp-variables, but not nat-variables, from a nat. However, exp is uncircumscribed by exp, whereas nat is not uncircumscribed by exp (because of the computational premise of omega). Thus, *weaken* allows for weakening an exp with an exp, but not with a nat (which would add a new case to the computational argument to omega). Moreover, the only rule bound by exp, namely exp, is uncircumscribed by exp, so *apply* allows for substituting an exp into an exp.

5 Programming Example

We present one simple example of mixing binding and computation; our companion technical report [22] contains several additional examples, including ones illustrating our approach to computing with open terms. We write the example using a named syntax for rule variables, which could

either be implemented directly in a proof assistant, or elaborated into de Bruijn form.

Consider the syntax of a simple language of arithmetic expressions, where numeric primitives are represented by computational functions. In LF, each primitive operation would require its own constructor; here, we represent binary primops (binops) uniformly as computational functions of type $\mathsf{nat} \otimes \mathsf{nat} \to \uparrow\mathsf{nat}$. The language includes numeric constants, binops, and let-binding:

$$\begin{array}{l}
\mathsf{zero} : \mathsf{nat},\ \mathsf{succ} : \mathsf{nat} \Leftarrow \mathsf{nat}, \\
\mathsf{num} : \mathsf{ari} \Leftarrow \mathsf{nat} \\
\mathsf{binop} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{nat} \otimes \mathsf{nat} \to \uparrow\mathsf{nat}) \Leftarrow \mathsf{ari} \\
\mathsf{let} : \mathsf{ari} \Leftarrow \mathsf{ari} \Leftarrow (\mathsf{ari} \Rightarrow \mathsf{ari})
\end{array}$$

For example, if *plus* is a function (negative value) of type $\langle\cdot\rangle\,(\mathsf{nat} \otimes \mathsf{nat} \to \uparrow\mathsf{nat})$ implementing addition on nats, then the value $(\mathsf{binop}\,(\mathsf{num}\,4)\,f\,(\mathsf{num}\,5))\,[\mathit{plus}/f]$ is the abstract syntax for the arithmetic expression "4 + 5". We implement an evaluator for closed programs using a fixed point:

$$\begin{array}{l}
\cdot \vdash \mathit{eval} : \langle\rangle\,\mathsf{ari} \to \uparrow\mathsf{nat} \\
\mathit{eval} = \mathsf{fix}(\mathit{ev}.\mathsf{val}^{-}(p\,; \varepsilon \mapsto \mathit{ev}^{*}\,p))
\end{array}$$

The body of the negative value is defined by a meta-function (an Agda function in our Agda implementation), where the

variable ev is the recursive reference:

$$
\begin{array}{l}
\forall(\Delta \Vdash c :: \langle\rangle\,\mathsf{ari}) : \;\; (ev : \langle\rangle\,\mathsf{ari} \to \uparrow\mathsf{nat}, \Delta) \vdash (ev^*\, c) : \langle\rangle\,\mathsf{nat} \\
ev^*\ \mathsf{num}\, p \qquad\qquad \mapsto \quad p \\
ev^*\ \mathsf{binop}\, p_1\, f\, p_2 \;\mapsto \\
\quad ev \bullet (p_1; \mathsf{cont}^+(p_1' \mapsto ev \bullet (p_2; \mathsf{cont}^+(p_2' \mapsto f \bullet (p_1', p_2')))) \\
ev^*\ \mathsf{let}\, p_0\ (\lambda\, u.p) \;\mapsto \\
\quad \mathit{apply} \bullet ((\lambda\, u.p, p_0); \mathsf{cont}^+(p' \mapsto ev \bullet p'))
\end{array}
$$

In this code, we employ a bit of syntactic sugar, suppressing the ε terminating a destructor pattern, the identity substitution id, and the identity case-analysis ε . The variables p are meta-variables ranging over patterns (Agda variables in our Agda implementation), which allow us to specify the behavior of ev^* on all arithmetic expressions using only finitely many cases. In the binop case, we recursively evaluate the first argument p_1, and match the result as p_1', then we evaluate the second argument p_2, and finally we apply the embedded computational function f to the values of the arguments. In the let case, we apply the body of the let to the let-bound term and evaluate the result. The function *apply*, which was discussed in Section 4, applies a representational function to an argument by performing substitution. The subordination conditions necessary for calling *apply* are satisfied in this

case: while the rules for ari use a computational function that circumscribes nat, they do not circumscribe ari.

6 Related Work

Variable binding can be implemented concretely in a variety of ways (see Aydemir et al. [5] for a survey). Among the concrete representation techniques, definitional variation is most similar to representations where the context of a term is marked in its type, such as de Bruijn representations using nested types or dependency [1, 6, 7]. In these representations, binders introduce a new constructor for variables, which are explicitly injected into terms. Our framework builds this use of dependency into the language: all types are contextual and all datatypes may be extended by rule variables introduced by $\Rightarrow$ and $\curlywedge$. This creates an opportunity to implement the structural properties once (modulo subordination conditions) for all types, including negative types such as computational functions, and to abstract away from the concrete implementation of variables themselves—as in LF, we can provide a named notation without requiring the programmer to manage names.

In systems based on the LF logical framework [17], LF

is taken as a pure representation language, and a separate layer is provided for computation. In Twelf [29], Delphin [33], and Beluga [30], the computational layer is an entirely separate language. Schürmann et al. [36] describe an approach in which the same arrow is used for both computation and representation, with primitive recursion isolated by a modality, but computation is nonetheless segregated because the computational modality cannot appear in rules. These stratified approaches have the advantage that all representations automatically obey the structural properties of a hypothetical judgement, with the disadvantage that certain encoding techniques, which rely on embedding computation in data, are not possible. Our framework removes this stratification, allowing rules that embed computation, with the consequence that not all representable rule systems satisfy the structural properties. However, as discussed in Section 4, we have implemented strengthening, weakening, and substitution generically under certain subordination conditions. Consequently, our framework provides meta-operations implementing the structural properties "for free" for all rule systems definable in simply-typed LF, as well as for many more rule systems that use iterated inductive definitions.

Our current framework lacks dependent types, a limita-

tion we plan to address in future work. In a dependently-typed setting, equality of terms influences equality of types, and equality of types influences type checking. In our setting, type checking will thus depend on the equational behavior of the meta-functions implementing the structural properties such as substitution. We are optimistic that the equational theory of the LF fragment of our framework will agree with LF definitional equality, even in an intensional setting, because we have implemented substitution by extending the hereditary substitution algorithm used in canonical LF [40]. However, we leave a detailed investigation of this issue to future work.

It is tempting to try to reuse the computational function space of existing proof assistants such as Coq and Isabelle/HOL to represent binding, but the naive approach admits too many functions. One solution to this problem is to use a predicate to identify those computational functions that are in fact substitution functions [2, 9, 11, 18, 26]. Another solution is to bind meta-language variables of an abstract type defined only by an axiomatic characterization of the properties of variables [8]. In contrast, our representational functions provide a direct means of adequately encoding binding, without requiring side conditions or axioms. Moreover, as we hope to have demonstrated, encod-

ing $\Rightarrow$ in terms of $\rightarrow$ ignores some of its essential properties, such as the distributivity principles in Section 2.5, and the ability to decompose a representational function by pattern-matching.

Nominal logic [13] is a theory of names and binding that has been implemented in several programming languages (e.g., FreshML [32, 37] and the Isabelle nominal datatype package [38]). The differences between the nominal approach and ours stem from the fact that FreshML separates fresh name generation from the binding of a name in a scope, whereas in our type theory rule variables do not exist outside of the scope in which they are bound. Nominal logic facilitates the direct representation of informal algorithms that use names without being explicit about their scope, whereas our approach follows the LF methodology of recasting these algorithms in terms of a more disciplined binding structure. Separating name generation from scoping makes it more difficult to determine what names are free in a computation, requiring freshness analyses [32], specification logics [34], or stateful operational semantics [37] in order to ensure that functions respect α-equivalence of representations. In contrast, the free rule variables of all computations are tracked by our type system, and respect for α-equivalence is achieved simply, by quotienting patterns

by α-equivalence.

In light of the present analysis, it is interesting to re-examine an old proposal by Miller [24] for an extension to ML with primitives for binding, including a new function type `'a => 'b` and a restricted form of higher-order pattern-matching. For example, the fact that the codomain `'b` must be an equality type in Miller's proposal is related to the present restriction that the codomain A^+ be positive—although it is less general, since A^+ can contain embedded negative formulas, which are not equality types. Technically, we are able to go beyond Miller's proposal because we associate negative hypotheses with a context of parameters. This idea appears in Miller and Tiu's more recent work [25], as well as in contextual modal type theory [27]. Indeed, Miller and Tiu's self-dual ∇ connective is closely related to $\Rightarrow$ and $\curlywedge$, also capturing the notion of a scoped constant. An essential difference, however, is that because the ∇ proof theory adopts a logic programming-based distinction between propositions and types (∇ quantifies over a type and forms a proposition), it is significantly less subtle than definitional variation. For example, ∇ cannot appear in the domain of a ∇ (in contrast to $\Rightarrow$).

Fiore et al. [12] and Hofmann [19] give semantic accounts of variable binding. It would be interesting to see

whether these semantic accounts can be extended to rule systems such as ours which permit computational functions in premises.

7 Conclusion

We have presented a language that enables the free interaction of binding with computation, extracted as the Curry-Howard interpretation of a focused sequent calculus with two forms of implication. We believe this provides an appropriate logical foundation, but much work remains to be done. We plan to pursue an independent implementation of our language by giving a first-order language for meta-functions, rather than relying on existing tools. Additionally, a generalization to dependent types (which we have already begun to explore [21]) would realize the goal of giving intrinsic support for variable binding in a constructive type theory, combining the best of frameworks such as Twelf and Coq.

Acknowledgements. We thank Frank Pfenning, Karl Crary, Jason Reed, Rob Simmons, Neel Krishnaswami, William Lovas, Arbob Ahmad, Brigitte Pientka, Ken Shan, and Andrew Pitts for helpful discussions about the work presented

here, and we thank our LICS reviewers for their thoughtful guidance on improving this article.

References

[1] T. Altenkirch and B. Reus. Monadic presentations of lambda terms using generalized inductive types. In *CSL 1999: Computer Science Logic*. LNCS, Springer-Verlag, 1999.

[2] S. Ambler, R. L. Crole, and A. Momigliano. Combining higher order abstract syntax with tactical theorem proving and (co)induction. In *International Conference on Theorem Proving in Higher-Order Logics*, pages 13–30, London, UK, 2002. Springer-Verlag.

[3] J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2(3):297–347, 1992.

[4] A. Avron. Simple consequence relations. *Information and Computation*, 92(1):105–139, 1991.

[5] B. Aydemir, A. Charguéraud, B. C. Pierce, R. Pollack, and S. Weirich. Engineering formal metatheory. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–15, 2008.

[6] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2–3):287–311, 1994.

[7] R. S. Bird and R. Paterson. De Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

[8] A. Bucalo, M. Hofmann, F. Honsell, M. Miculan, and I. Scagnetto. Consistency of the theory of contexts. *Journal of Functional Programming*, 16(3):327–395, May 2006.

[9] V. Capretta and A. Felty. Combining de Bruijn indices and higher-order abstract syntax in Coq. In *Proceedings of TYPES 2006*, volume 4502 of *Lecture Notes in Computer Science*, pages 63–77. Springer-Verlag, 2007.

[10] Coq Development Team. *The Coq Proof Assistant Reference Manual*. INRIA, 2007. Available from `http://coq.inria.fr/`.

[11] J. Despeyroux, A. Felty, and A. Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *International Conference on Typed Lambda Calculi and Applications*, volume 902 of *Lecture Notes in Computer Science*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

[12] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *IEEE Symposium on Logic in Computer Science*, 1999.

[13] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax involving binders. In *IEEE Symposium on Logic in Computer Science*, pages 214–224. IEEE Press, 1999.

[14] J.-Y. Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

[15] J.-Y. Girard. On the unity of logic. *Annals of pure and applied logic*, 59(3):201–217, 1993.

[16] L. Hallnäs. Partial inductive definitions. *Theoretical Com-*

puter Science, 87(1):115–142, 16 Sept. 1991.

[17] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1), 1993.

[18] J. Hickey, A. Nogin, X. Yu, and A. Kopylov. Mechanized meta-reasoning using a hybrid HOAS/de Bruijn representation and reflection. In *ACM SIGPLAN International Conference on Functional Programming*, pages 172–183, New York, NY, USA, 2006. ACM.

[19] M. Hofmann. Semantical analysis of higher-order abstract syntax. In *IEEE Symposium on Logic in Computer Science*, 1999.

[20] D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *IEEE Symposium on Logic in Computer Science*, pages 162–172. IEEE Computer Society, 1991.

[21] D. R. Licata and R. Harper. Dependently typed programming with domain-specific logics. Draft available from `http://www.cs.cmu.edu/~drl`, April 2008.

[22] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. Technical Report CMU–CS–08–101, Carnegie Mellon University School of Computer Science, Pittsburgh, PA, April 2008.

[23] P. Martin-Löf. *Hauptsatz* for the intuitionistic theory of iterated inductive definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216, Amsterdam, 1971. North Holland.

[24] D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. Technical report, Penn-

sylvania State University, Department of Computer Science and Engineering, Aug. 1990.

[25] D. Miller and A. F. Tiu. A proof theory for generic judgments: An extended abstract. In *IEEE Symposium on Logic in Computer Science*, pages 118–127, 2003.

[26] A. Momigliano, A. Martin, and A. Felty. Two-level hybrid: A system for reasoning using higher-order abstract syntax. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2007.

[27] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

[28] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers University of Technology, 2007.

[29] F. Pfenning and C. Schürmann. System description: Twelf - a meta-logical framework for deductive systems. In H. Ganzinger, editor, *International Conference on Automated Deduction*, pages 202–206, 1999.

[30] B. Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 371–382, 2008.

[31] A. M. Pitts. Nominal logic, a first order theory of names and binding. *Information and Computation*, 186:165–193, 2003.

[32] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Sci-*

ence, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[33] A. Poswolsky and C. Schürmann. Practical programming with higher-order encodings and dependent types. In *European Symposium on Programming*, 2008.

[34] F. Pottier. Static name control for FreshML. In *IEEE Symposium on Logic in Computer Science*, 2007.

[35] P. Schroeder-Heister. Rules of definitional reflection. In R. L. Constable, editor, *IEEE Symposium on Logic in Computer Science*, pages 222–232, Montreal, Canada, June 1993. IEEE Computer Society Press.

[36] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266:1–57, 2001.

[37] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: Programming with binders made simple. In *ACM SIGPLAN International Conference on Functional Programming*, pages 263–274, August 2003.

[38] C. Urban. Nominal techniques in Isabelle/HOL. *Journal of Automatic Reasoning*, 2008. To appear.

[39] R. Virga. *Higher-Order Rewriting with Dependent Types*. PhD thesis, Carnegie Mellon University, 1999.

[40] K. Watkins, I. Cervesato, F. Pfenning, and D. Walker. A concurrent logical framework I: Judgments and properties. Technical Report CMU-CS-02-101, Department of Computer Science, Carnegie Mellon University, 2002. Revised May 2003.

[41] N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008. Special issue on "Classical

Logic and Computation".

[42] N. Zeilberger. Focusing and higher-order abstract syntax. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369, 2008.

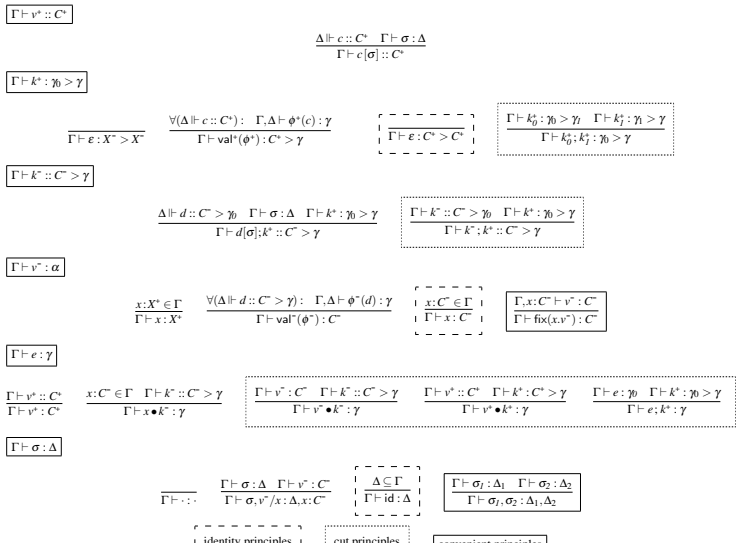
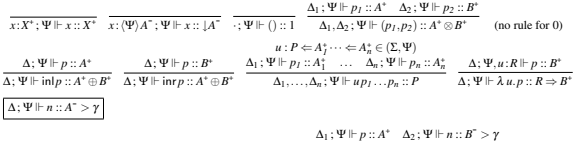


Figure 6. Focusing rules with proof terms



$$\frac{}{\cdot\,;\Psi \Vdash \varepsilon :: X^- > X^-} \qquad \frac{}{\cdot\,;\Psi \Vdash \varepsilon :: \uparrow A^+ > \langle\Psi\rangle A^+} \qquad \frac{}{\Delta_1,\Delta_2\,;\Psi \Vdash p\,;n :: A^+ \to B^- > \gamma}$$

$$\frac{\Delta\,;\Psi \Vdash n :: A^- > \gamma}{\Delta\,;\Psi \Vdash \mathsf{fst};n :: A^- \& B^- > \gamma} \qquad \frac{\Delta\,;\Psi \Vdash n :: B^- > \gamma}{\Delta\,;\Psi \Vdash \mathsf{snd};n :: A^- \& B^- > \gamma} \qquad \frac{\Delta\,;\Psi, u:R \Vdash n :: B^- > \gamma}{\Delta\,;\Psi \Vdash \mathsf{unpack};u.n :: R \wedge B^- > \gamma}$$

(no rule for $\top$)

$$\boxed{\Delta \Vdash c :: \langle\Psi\rangle A^+ \text{ and } \Delta \Vdash d :: \langle\Psi\rangle A^+ > \gamma}$$

$$\frac{\Delta\,;\Psi \Vdash p :: A^+}{\Delta \Vdash \overline{\Psi}.p :: \langle\Psi\rangle A^+} \qquad \frac{\Delta\,;\Psi \Vdash n :: A^- > \gamma}{\Delta \Vdash \overline{\Psi}.n :: \langle\Psi\rangle A^- > \gamma}$$

Figure 7. Constructor and destructor patterns