Focusing on Binding and Computation

Daniel R. Licata Noam Zeilberger Robert Harper

February, 2008
CMU-CS-08-101

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

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 the notion of a scoped inference rule, which permits the adequate representation of binding via higher-order abstract syntax. On the other hand, the usual *computational arrow* classifies recursive functions over such higher-order data. Unlike many previous approaches, both binding and computation can mix freely. Following Zeilberger [POPL 2008], the computational function space admits a form of open-endedness, in that it is represented by an abstract map from patterns to expressions. As we demonstrate with Coq and Agda implementations, this has the important practical benefit that we can reuse the pattern coverage checking of these tools.

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.

Keywords: type theory, proof theory, focusing, polarity, variable binding, higher-order abstract syntax, definitional reflection

1 Introduction

A logical framework provides a set of abstractions that facilitate the representation of *logical systems* such as programming languages and logics. Moreover, logical frameworks enable generic infrastructure, such as tools for computing with and reasoning about logical systems, to be implemented once for a framework and reused across many logical systems. In this sense, one of the most well-known logical frameworks is the datatype mechanism of functional programming languages such as SML and Coq [Milner et al., 1997, Coq Development Team, 2007]. Datatypes permit facile representations of the first-order algebraic terms that feature prominently in the syntax and proof theory of logical systems, and the ambient functional programming language provides a generic mechanism for computing with such representations. The LF logical framework [Harper et al., 1993] enriches ordinary datatypes with one additional abstraction: intrinsic support for binding and scope, facilitating the representation of α-convertible variables, capture-avoiding substitution, and hypothetical judgements (uniform reasoning from assumptions to conclusions). However, LF is only a representation language, and requires a separate language, such as Twelf [Pfenning and Schürmann, 1999], for computation. Because of this stratification, LF is less general than ML datatypes in one important way: it is impossible to embed computations in data. There is no LF analogue of a datatype with an SML function as a component, nor of an iterated inductive definition [Martin-Löf, 1971]. In this work, we focus on providing an abstraction for representing binding, as in LF, while simultaneously allowing computations to be embedded in data, as in ML.

Our approach to this problem builds on definitional reflection [Schroeder-Heister, 1984], which is a logical analogue of ML datatypes. Definitional reflection internalizes definition by inference rules as a logical primitive: a logic with definitional reflection contains a class of atoms P defined by a database Ψ of inference rules, which have the form $P \Leftarrow A_1 \Leftarrow \ldots \Leftarrow A_n$. For example, the two rules $(P \Leftarrow A_1 \, ; \, P \Leftarrow A_2 \Leftarrow A_3)$ define P as equivalent to $A_1 \vee (A_2 \wedge A_3)$. Recursive definition is also allowed; for example, the rule database $\Psi_{\mathbb{N}} = (\mathsf{nat} \, ; \, \mathsf{nat} \Leftarrow \mathsf{nat})$ defines the natural numbers. The sequent calculus right rule for defined atoms is rule application:

$$\frac{P \Leftarrow A_1 \Leftarrow \ldots \Leftarrow A_n \in \Psi \quad \langle\Psi\rangle\, \Gamma \Longrightarrow A_1 \quad \cdots \quad \langle\Psi\rangle\, \Gamma \Longrightarrow A_n}{\langle\Psi\rangle\, \Gamma \Longrightarrow P}$$

The left rule for defined atoms is the *definitional reflection* rule[1]:

$$\frac{\forall (P \Leftarrow A_1 \Leftarrow \ldots \Leftarrow A_n \in \Psi) : \quad \langle\Psi\rangle\, \Gamma, A_1, \ldots, A_n \Longrightarrow C}{\langle\Psi\rangle\, \Gamma, P \Longrightarrow C}$$

The definitional reflection rule can be read as asserting that the rule database is exhaustive: to reason from P, it suffices to reason from the premises of all rules concluding P. Through the Curry-Howard correspondence, we can think of inference rules as datatype *constructors*; e.g., the two rules in $\Psi_{\mathbb{N}}$ correspond to zero and successor. The right rule for P builds a term by applying a datatype constructor. The left rule states that P may be destructed by *pattern-matching*, giving a case for each possible constructor.

Now, there turns out to be a close relationship between the ability to destruct a logical constant by pattern-matching and the logical notions of *focusing* and *polarity* [Andreoli, 1992, Girard, 1993, Zeilberger, 2008]. Roughly, only the constructors for *positive* connectives (like multiplicative conjunction $\otimes$ and additive disjunction $\oplus$ in linear logic) can be matched against. Connectives of *negative* polarity (like additive conjunction $\&$ and implication $\multimap$ in linear logic) must have explicit destructors (e.g., first and second projections, application), though there is a dual sense in which these may be matched against to *construct* a term.

[1] I.e., the rule "reflects" upon the possible proofs of P according to its definition.

Our work began by observing a seeming paradox about the polarity of variable binding: In some respects, binding behaves like implication, ordinarily negative, with application defined by substitution. However, it is also possible to pattern-match against data with variable binding, suggesting it may correspond to a positive connective.

In this paper, we propose an extension of definitional reflection called *definitional variation*, which, via Curry-Howard, yields a functional programming language with intrinsic support for representing and computing with binding. We present this logic as a focused sequent calculus in which the definitions of atoms are open-ended, and can be varied by means of logical connectives. One such connective is indeed a positive form of implication $R \Rightarrow A$, called the *representational arrow*. A proof of $R \Rightarrow A$ is a proof of A which may use the additional inference rule R: this connective introduces a new, scoped inference rule, which corresponds to a new, scoped datatype constructor. The representational arrow thus provides an abstraction for encoding binding through higher-order abstract syntax. On the other hand, the familiar *computational* arrow $\rightarrow$, of negative polarity, classifies clausal, pattern-matching functions over such higher-order data. This yields a programming language which has the expressiveness of ML (via the computational arrow) while permitting direct representations of variable binding (via the representational arrow). Moreover, both function spaces can be used in datatype definitions, providing datatypes that freely mix binding and computation. We leave the study of a dependent version of this language, which would enrich a type theory such as Coq with intrinsic support for binding and scope, to future work.

Following Zeilberger [2008], our computational arrow admits a form of open-endedness, in that such functions are represented abstractly by meta-level functions from patterns to expressions. These meta-level functions can be taken to be constructive, in which case all implications are effectively computable, or non-constructive, in which case the operational behavior is necessarily oracular (as in Howe [1991]). The open-endedness of the computational arrow has important practical benefits. First, these meta-level functions can be presented as programs in existing proof assistants, which permits us to reuse existing pattern coverage checkers. Second, open-endedness means that functions written in several proof assistants, using different implementations of binding, can be combined in a single program.

The technical contributions of this paper are as follows:

- In Section 2, we present a focused sequent calculus for definitional variation, and give the expansions and reductions witnessing admissibility of identity and cut.
- In Section 3, we discuss some interesting distributivity properties of $\Rightarrow$, and prove that the type theory includes (simply-typed) LF as a subsystem—proving that the representational arrow adequately represents binding.
- In Section 4, we give a proof term assignment to our sequent calculus, yielding a functional programming language with an operational semantics given by cut elimination. We prove type safety, and illustrate programming in this language via several examples that mix binding and computation. Additionally, we discuss implementations of the type theory in Agda [Norell, 2007] and Coq [Bertot and Castéran, 2004].

Comparison with Other Techniques for Representing Binding

We now describe the high-level differences between our approach to representing binding and other techniques which have been discussed in the literature. We defer a detailed, technical comparison with related work to Section 5.

Concrete Implementations One approach to computing with binding is simply to work directly with a concrete implementation. The most minimalist implementation of binding is de Bruijn indices [de Bruijn, 1972], in which variables are represented as numbers indexing positionally into a context. However, de Bruijn indices are regarded as being difficult for programmers to work with: for example, terms are thought to be difficult to write and to read, and structural properties such as weakening (extending a context with additional variables) and exchange (permuting variables in the context) require indices to be adjusted. Consequently, it is common to represent the indices into the context as some named form of atom (e.g., strings). Unfortunately, when bound variables are represented by atoms, a given bound variable can be represented equally well by any atom, so terms must be explicitly quotiented by α-equivalence. When free variables are represented by atoms, inference rules must be carefully crafted to ensure that they yield general enough inductive hypotheses for proving theorems by rule induction (see [Aydemir et al., 2008] for comparison of exists-fresh, forall-fresh, and cofinite rules). An alternative to choosing between de Bruijn and named form is to use one representation for bound variables and another for free variables. For example, locally nameless (de Bruijn indices for bound variables) / globally named (names for free variables) is thought to provide a good balance of advantages and disadvantages [Aydemir et al., 2008].

To our minds, the chief disadvantage of working directly with a concrete implementation of binding is that it provides too "leaky" an interface. That is, the programmer is exposed to the gory details of the particular representation of binding, and must program differently for different representations. Moreover, it may not be easy to port code written with respect to one implementation of binding to another.

In contrast to these concrete implementations, our representational function space provides an abstract interface for binding. Variables in a logical system of interest (an *object language*) are represented by inference rule variables in our type theory. This interface hides the concrete details of how these variables are implemented, and, we argue, leads to natural representations and code. Moreover, by exploiting the open-endedness of our computational function space, programmers who wish to work with a particular concrete implementation of binding may do so.

Nominal Logic Unlike the above concrete implementations, nominal logic [Gabbay and Pitts, 1999] does provide an abstract interface for binding, and this interface has been implemented in several programming languages (e.g., FreshML [Pitts and Gabbay, 2000, Shinwell et al., 2003] and the Isabelle nominal datatype package [Urban, 2008]). In nominal logic, variables of a logical system are represented using names a, and binders are represented by a primitive $<a>\ t$, which pairs a name a with a term t. The nominal logic interface for binding then provides notions of name permutation and freshness, α-equivalence for binders, and induction and recursion modulo α-equivalence. This interface hides the details of these operations from programmers, and provides a reusable implementation of binding.

However, the nominal apparatus is, in a sense, more general than what is required simply to represent variable binding. This is because names in nominal logic are atoms with *global scope*: it is always permissible to mention a name a (just as it is always permissible to mention a string in ML) even if there is no enclosing binder bringing a into scope. Consequently, the nominal interface for binding requires machinery for ensuring that certain names are fresh with respect to (roughly "not free in") certain computations. For example, in order for a function to be well-defined on α-equivalence classes of nominal terms, it is necessary to prove that the result of the function is independent of the name a appearing in any binder $<a>\ t$ that the function processes—i.e., that the name appearing in the binder is fresh with respect to the result of the function. This *freshness condition for binders* [Pitts, 2006] must be proved for each function definition. Pitts and Gabbay [2000] employ a conservative freshness analysis to discharge these conditions. Pottier [2007] describes a specification logic in which these conditions can be proved. Shinwell et al. [2003] exploit an effectful operational semantics to ensure that the conditions cannot be violated.

In contrast, we present an interface for binding in which variables are inherently scoped: a representational function introduces a new, scoped constant, notated by a meta-level variable. Just as an ill-scoped ML program will be rejected by the type checker, an ill-scoped object-language program will not be representable as data using this interface for binding. Moreover, it is impossible for a function to violate the freshness condition for binders: since a binder in our interface is *not* a pair of a name (which is a piece of data) and a term, but a meta-level binder of a variable in a term, there is no name to be dependent on. Our type theory ensures respect for α-equivalence using little more infrastructure than is necessary to scope-check an ML program. Nonetheless, when they are in scope, variables are treated just like any other datatype constructor, and therefore can be pattern-matched against, compared for equality, etc, permitting a programming style similar to nominal code.

Higher-Order Abstract Syntax In the LF logical framework [Harper et al., 1993], variable binding is represented using *higher-order abstract syntax* (HOAS): object language variables are represented by LF variables, and object language binders are represented by the LF function space. Object-language α-conversion is represented by LF α-conversion, and object-language substitution is represented by LF substitution. Such substitutions arise from *applying* an LF function representing a binder, using a standard implication elimination rule (modus ponens). These techniques adequately capture the notion of variable binding because object-language entities are represented by uninterpreted base types (not inductive types) in LF; consequently, an LF function cannot case-analyze its argument, but must use it uniformly. Thus, the LF function space provides an adequate interface for representation, but no ability for computation. Computational languages such as Twelf [Pfenning and Schürmann, 1999] provide an additional layer on top of LF, in which the terms of the LF type theory itself are treated as an inductive definition. For example, in Twelf, Π_2 metatheorems about LF terms are proved by using the induction principle for LF terms to define total relations. This stratification explains why the LF function space seems to have two different elimination forms: in LF, a function is eliminated by application, which gives substitution; in Twelf, where the terms of LF are treated as an inductive datatype, it is eliminated by pattern-matching. As we noted above, this stratification also means that there is no way to embed a Twelf computation in an LF representation.

In this paper, we avoid the stratified approach taken by LF/Twelf, as both our representational arrow (which provides the functionality of the LF function space) and our computational arrow (which provides the functionality of Twelf metatheorems) are connectives in a single language. However, this means we must choose sides: is the representational arrow eliminated by application (as in LF) or by pattern-matching (as in Twelf)? Here, we take the Twelf elimination form as primitive: our representational arrow $R \Rightarrow A$ is eliminated by pattern-matching, which exposes a value of type A potentially using the rule R. Consequently, the representational arrow is a funny sort of implication, in that it need not satisfy modus ponens: we need not make an *a priori* commitment to a substitution principle for rules R in the rule context. In contrast, because LF functions are internally eliminated by modus ponens, LF representations inherently commit to a notion of substitution for object-language variables. Our representational implication $R \Rightarrow A$ thus provides primitive support for *scoped constants*, without committing to structural properties, such as substitution, for the rule context. Indeed, in our type theory, it is not necessarily the case that these structural properties hold for all rule systems, because computational functions can be used in the premises of rules. Such rules have proven quite useful in frameworks, such as Coq, based on iterated inductive definitions [Martin-Löf, 1971, Coquand and Paulin-Mohring, 1989]; one common use is to negate an inductive definition, with a premise $P \to \bot$ asserting the refutability of P. However, such rules can invalidate structural properties: for example, given a derivation including a proof of $\langle\Psi\rangle\ P \to \bot$, it is not possible to weaken Ψ by adding P.

Thus, our language of inference rules is more general than LF, in that it permits computational premises. In compensation, this generalization weakens what can be said generically about the class of all rule systems that can be represented. However, this weakening is not an obstacle in practice: we prove that structural properties such as substitution hold generically for all LF-like rule systems, recovering the benefits of the LF elimination form for its function space. By exploiting computational open-endedness, we may implement this proof as a datatype-generic program, so that when programmers restrict themselves to LF-style representations, they can rely on the rule context behaving like a hypothetical judgement. However, if a programmer uses more general representation techniques, such as iterated inductive definitions, the structural properties of the hypothetical judgement are not assured, and explicit justification must be provided. In this sense our calculus unifies the practical benefits of the LF approach, where the structural properties are available "for free", with the benefits of more concrete approaches, which permit more general representation techniques at the expense of demanding proof of the structural properties.

2 Focusing on Definitional Variation

In this section, we present a focused sequent calculus for intuitionistic definitional variation. When describing this calculus, we foreshadow the proof-term assignment given in Section 4, freely interchanging logical and type-theoretic terminology ("proposition" and "type", "implication" and "function space", "logic" and "type theory", etc.).

Before discussing the technical details, we build intuition for the polarities of the connectives. One way to view *positive* and *negative* polarity is in terms of Michael Dummett's distinction between verificationist and pragmatist "meaning-theories" [Dummett, 1991]. Positive connectives are in a sense "defined" by how you verify them; the sequent calculus we present makes this idea formal, in that a positive connective is fully specified by axiomatizing the structure of the values that introduce it. Consequently, its elimination form is any context that consumes all such values. Dually, negative connectives are "defined" by how you use them; in our presentation, a negative connective is fully specified by the observations its elimination forms make. Consequently, its introduction form is any data that supports all such observations.

For example, the computational arrow is negative because it is biased towards use: to observe a function of type $A \to B$, apply it to any value of type A and then observe the result B. Consequently, a computational arrow may be introduced by giving a proof of B for every value of type A—in other words, by defining a function using pattern-matching. On the other hand, the representational arrow is positive because it is biased towards verification: to build a value of type $R \Rightarrow A$, build a value of type A using the additional inference rule R. Consequently, a representational arrow is *eliminated* by pattern-matching, corresponding to informal proofs by rule induction on syntax with variable binding.

Our focused sequent calculus is defined in two stages, following the style of Zeilberger [2007]. First, the (polarized) connectives are defined by axiomatizing the structure of (positive) values and (negative) observations. Second, there is a general focusing framework that is independent of the particular connectives of the logic. The two judgements defining the connectives are conceptually prior, in that the remaining judgements quantify over them. For the sake of presentation, we start by describing the connective judgments in the simple propositional case, then present the general focusing rules, and finally return to revise the definitions of the connectives in the setting of definitional variation.

We write C^+ and C^- to stand for positive and negative formula, X^+ and X^- for positive and negative propositional variables (atomic propositions), and Δ to stand for a list of negative formulas and positive atoms. The positive connectives are defined using the judgement $\Delta \Vdash C^+$, which corresponds to applying only *linear right-rules* to show C^+ from Δ. For example, the rules defining conjunction, disjunction, and atoms are as follows:

$$\frac{}{X^+ \Vdash X^+} \qquad \frac{\Delta_1 \Vdash C_1^+ \quad \Delta_2 \Vdash C_2^+}{\Delta_1, \Delta_2 \Vdash C_1^+ \otimes C_2^+} \qquad \frac{\Delta \Vdash C_1^+}{\Delta \Vdash C_1^+ \oplus C_2^+} \qquad \frac{\Delta \Vdash C_2^+}{\Delta \Vdash C_1^+ \oplus C_2^+}$$

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

Negative connectives are defined using the judgement $\Delta; C^- \Vdash \gamma$, which corresponds to using *linear left-rules* to decompose C^- into the consequence γ, which is either a negative atom X^- or a positive formula C^+. The derivations are elimination contexts for negative types (which are ordinarily called *spines*), except that the contexts are made up of patterns rather than full terms; hence we refer to them as *spine patterns.* The rules for atoms, ordinary implication, and negative conjunction are as follows:

$$\frac{}{\cdot; X^- \Vdash X^-} \qquad \frac{\Delta_1 \Vdash C_1^+ \quad \Delta_2; C_2^- \Vdash \gamma}{\Delta_1, \Delta_2; C_1^+ \to C_2^- \Vdash \gamma} \qquad \frac{\Delta; C_1^- \Vdash \gamma}{\Delta; C_1^- \& C_2^- \Vdash \gamma} \qquad \frac{\Delta; C_2^- \Vdash \gamma}{\Delta; C_1^- \& C_2^- \Vdash \gamma}$$

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 different 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 constructing a lazy pair.

2.1 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., as in ML, 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 *values*: a value is a value 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 value pattern, and then satisfy the pattern's free variables. Values are eliminated the left-inversion judgement $\Gamma; \gamma_0 \vdash \gamma$, which defines a *value match*, or case-analysis. Inversion steps respond to all possible choices that the corresponding focus step can make: the rule for C^+ quantifies over all value 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 $\Gamma, \Delta \vdash \gamma$." Here γ ranges over consequences, which are either negative atoms or positive formulae; for atoms, the only case-analysis is the identity. The positive connectives are thus introduced by choosing a value (focus) and eliminated by contexts that are prepared to handle any such value (inversion).

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 $\Gamma; [C^-] \vdash \gamma$, which chooses how to observe C^- by giving a spine. A spine consists of a spine pattern, a substitution, and a case-analysis. The spine 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

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

Right Focus $\boxed{\Gamma \vdash [C^+]}$

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

Left Inversion $\boxed{\Gamma; \gamma_0 \vdash \gamma}$

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

Left Focus $\boxed{\Gamma; [C^-] \vdash \gamma}$

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

Right Inversion $\boxed{\Gamma \vdash \alpha}$

$$\frac{\forall(\Delta; C^- \Vdash \gamma) : \quad \Gamma, \Delta \vdash \gamma}{\Gamma \vdash C^-} \qquad \frac{X^+ \in \Gamma}{\Gamma \vdash X^+}$$

No Focus $\boxed{\Gamma \vdash \gamma}$

$$\frac{\Gamma \vdash [C^+]}{\Gamma \vdash C^+} \qquad \frac{C^- \in \Gamma \quad \Gamma; [C^-] \vdash \gamma}{\Gamma \vdash \gamma}$$

Assumptions $\boxed{\Gamma \vdash \Delta}$

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta \quad \Gamma \vdash \alpha}{\Gamma \vdash \Delta, \alpha}$$

Figure 1: Focusing rules

judgement $\Gamma \vdash \alpha$, where assumptions α are negative formula or positive atoms, specifies the structure of a *spine match*. A spine match for C^- must show that for all spine patterns decomposing C^-, the conclusion of the spine pattern is justified by the variables bound by the patterns in it.

The judgement $\Gamma \vdash \gamma$, defines an unfocused sequent, or an *expression*: from an expression, one can right-focus and introduce a value, or left-focus on an assumption in Γ and apply a spine to it. Finally, a *substitution* $\Gamma \vdash \Delta$ provides a spine-match 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:

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

2.2 Patterns for Definitional Variation

Figure 2 defines the syntax of propositions, along with the value and spine pattern judgements fixing their meaning. Modulo notation, most of the connectives are standard from polarized logic [Girard, 2001, Laurent, 2002, Liang and Miller, 2007, Zeilberger, 2007]: positive formulae (A^+) include positive atoms (X^+), nullary and binary products and sums (1, $A^+ \otimes B^+$, 0, $A^+ \oplus B^+$), and shifted negative formulae ($\downarrow A^-$); negative formulae (A^-) include negative atoms (X^-), computational implication ($A^+ \to B^-$), negative nullary and binary conjunction ($\top$ and $A^- \& B^-$), and shifted positive formulae ($\uparrow A^+$). Additionally, we introduce positive defined atoms (P), defined by an open-ended collection of rules. Inference rules (R) take the form $P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+$ ("conclude P given proofs of A_1^+ through A_n^+"), and are collected in the rule context Ψ. Above, we took C^+ and C^- to range over polarized formulae, but in this more general setting they range over *contextual* polarized formulae $\langle\Psi\rangle\, A^+$ and $\langle\Psi\rangle\, A^-$.

Intuitively, a derivation of $\Delta \Vdash \langle\Psi\rangle\, A^+$ represents a value pattern of type A^+ using the constructors in Ψ an arbitrary number of times, and naming the variables in Δ linearly. Since values of negative types cannot be decomposed, as a base case we have $\langle\Psi\rangle\, A^- \Vdash \langle\Psi\rangle \downarrow A^-$ corresponding to a variable pattern. Likewise, spines of positive type cannot be decomposed, so that we have $\cdot; \langle\Psi\rangle \uparrow A^+ \Vdash \langle\Psi\rangle\, A^+$ as a base case. Note that rule contexts are not associated with atomic hypotheses X^+ or consequences X^-.

The truly exotic connectives are $\Rightarrow$ (of positive polarity) and $\curlywedge$ (of negative polarity), which manipulate the rule context. The meanings of these connectives are explained as follows: To build a value of type $R \Rightarrow A^+$, extend the rule context with R and build a value of type A^+. To observe $R \curlywedge A^-$, extend the rule context with R and observe A^-. Note that in both cases, the new rules can eventually find their way into contextual hypotheses or conclusions. Ignoring structural punctuation, the definition of $\Rightarrow$ is simply the usual implication right-rule, while the definition of $\curlywedge$ is simply a conjunction left-rule. However, as we will see in Section 3.1, in many ways these connectives behave quite differently from "ordinary" implication and conjunction.

Taken together, the pattern rule for defined atoms P, which introduces P by application of a rule in Ψ, and the rules for right- and left- inversion, which universally quantify over all such introductions, yield the definitional reflection rule cited in Section 1. E.g., for $\Psi = \langle P \Leftarrow X^+ \,;\, P \Leftarrow Y^+ \Leftarrow Z^+\rangle$, the following rule is derivable:

$$\frac{\Gamma, X^+ \vdash \gamma \quad \Gamma, Y^+, Z^+ \vdash \gamma}{\Gamma; \langle\Psi\rangle\, P \vdash \gamma}$$

However, whereas the traditional definitional reflection rule performs only a single step of unrolling a definition, the focusing inversion rule unrolls definitions until they reach a polarity switch, which in some

Pos. formula	A^+	::=	$X^+ \mid {\downarrow}A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+ \mid P \mid R \Rightarrow A^+$
Rule	R	::=	$P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+$
Neg. formula	A^-	::=	$X^- \mid {\uparrow}A^+ \mid A^+ \rightarrow B^- \mid \top \mid A^- \& B^- \mid R \curlywedge B^-$
Rule Context	Ψ	::=	$\cdot \mid \Psi, R$
CPF	C^+	::=	$\langle\Psi\rangle\, A^+$
CNF	C^-	::=	$\langle\Psi\rangle\, A^-$

$\boxed{\Delta \Vdash \langle\Psi\rangle\, A^+}$

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

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

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

$$\frac{\Delta \Vdash \langle\Psi, R\rangle\, B^+}{\Delta \Vdash \langle\Psi\rangle\, R \Rightarrow B^+} \qquad \frac{\begin{array}{c}P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+ \in \Psi \\ \Delta_1 \Vdash \langle\Psi\rangle\, A_1^+ \quad \ldots \quad \Delta_n \Vdash \langle\Psi\rangle\, A_n^+\end{array}}{\Delta_1, \ldots, \Delta_n \Vdash \langle\Psi\rangle\, P}$$

$\boxed{\Delta; \langle\Psi\rangle\, A^- \Vdash \gamma}$

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

$$\frac{\Delta_1 \Vdash \langle\Psi\rangle\, A^+ \quad \Delta_2; \langle\Psi\rangle\, B^- \Vdash \gamma}{\Delta_1, \Delta_2; \langle\Psi\rangle\, A^+ \rightarrow B^- \Vdash \gamma}$$

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

$$\frac{\Delta; \langle\Psi, R\rangle\, B^- \Vdash \gamma}{\Delta; \langle\Psi\rangle\, R \curlywedge B^- \Vdash \gamma}$$

Figure 2: Value and spine patterns

cases gives an ω-rule. For example, for the rule context $\Psi_{\mathbb{N}}$ defined in the introduction, a derivation of $\Gamma; \langle\Psi_{\mathbb{N}}\rangle\, \mathsf{nat} \vdash \gamma$ has one premise for each natural number.

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:

$$\begin{array}{l} \mathsf{lam} : \mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp}) \\ \mathsf{app} : \mathsf{exp} \Leftarrow \mathsf{exp} \Leftarrow \mathsf{exp} \end{array}$$

For clarity, we name the rules in the rule context here, foreshadowing the presentation with proof terms in Section 4. The λ-calculus terms with free variables $x_1, \ldots, x_n$ are isomorphic to derivations of the value pattern judgement $\cdot \Vdash \langle \Psi_\lambda, x_1 : \mathsf{exp}, \ldots, x_n : \mathsf{exp}\rangle\, \mathsf{exp}$. 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 such higher-order data. E.g., a term $\cdot \vdash \langle \Psi_\lambda \rangle\, \mathsf{exp} \rightarrow \uparrow\!\mathsf{exp}$ represents a function from λ-terms in the empty context to λ-terms in the empty context.

2.3 Identity and Cut

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

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

The type D defined by this constant is essentially the recursive type $\mu D.D \rightarrow D$, 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 principles, "prove" them by operationally sound but possibly non-terminating arguments, and then discuss criteria under which these proofs are well-founded.

Principle 1 (Identity)**.**

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

Procedure. The first identity principle reduces to the second and third as follows:

$$\dfrac{\forall(\Delta; C^- \Vdash \gamma) : \quad \dfrac{C^- \in \Gamma \qquad \dfrac{\Delta; C^- \Vdash \gamma \qquad \overset{\mathsf{ID3}}{\Gamma, \Delta \vdash \Delta} \qquad \overset{\mathsf{ID2}}{\Gamma; \gamma \vdash \gamma}}{\Gamma, \Delta; [C^-] \vdash \gamma}}{\Gamma, \Delta \vdash \gamma}}{\Gamma \vdash C^-}$$

The second identity reduces to the third as follows:

$$\dfrac{\forall(\Delta \Vdash C^+) : \quad \dfrac{\dfrac{\Delta \Vdash C^+ \quad \overset{\mathsf{ID3}}{\Gamma, \Delta \vdash \Delta}}{\Gamma, \Delta \vdash [C^+]}}{\Gamma, \Delta \vdash C^+}}{\Gamma; C^+ \vdash C^+}$$

Finally, the third identity reduces to the first applied over all hypotheses $C^- \in \Delta$.

□

Principle 2 (Cut)**.**

1. *(neg. reduction) If* $\Gamma \vdash C^-$ *and* $\Gamma; [C^-] \vdash \gamma$ *then* $\Gamma \vdash \gamma$.
2. *(pos. reduction) If* $\Gamma \vdash [C^+]$ *and* $\Gamma; C^+ \vdash \gamma$ *then* $\Gamma \vdash \gamma$.
3. *(composition)*
   - *(a) If* $\Gamma \vdash \gamma_0$ *and* $\Gamma; \gamma_0 \vdash \gamma$ *then* $\Gamma \vdash \gamma$.
   - *(b) If* $\Gamma; [C^-] \vdash \gamma_0$ *and* $\Gamma; \gamma_0 \vdash \gamma$ *then* $\Gamma; [C^-] \vdash \gamma$.
   - *(c) If* $\Gamma; \gamma_1 \vdash \gamma_0$ *and* $\Gamma; \gamma_0 \vdash \gamma$ *then* $\Gamma; \gamma_1 \vdash \gamma$.
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:

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

By plugging $\Delta; C^- \Vdash \gamma_0$ 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 $\Gamma; \gamma_0 \vdash \gamma$. 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 the above procedures make no mention of particular connectives or rule contexts, instead reasoning uniformly about focusing derivations. As we alluded to above, however, in general these procedures are 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* $C_1 > C_2$ *between contextual formulas as the least relation closed under transitivity and the following properties:*

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

For any contextual formula C*, we define* $>_C$ *to be the restriction of* $>$ *to formulas below* C*.*

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 *is well-founded if* $>_C$ *is well-founded.*

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

Proof. By inspection of the above procedure. Positive and negative identity are proved by mutual induction using the order $>_C$, with a side induction on the length of Δ to show substitution identity. $\square$

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 $>_C$, with a side induction on the left derivation to show composition, and a side induction on the right derivation to show substitution. $\square$

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* $\mathsf{D} \Leftarrow \downarrow(\mathsf{D} \to \uparrow D)$ *is not.*

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

Proof. By induction on the structure of A. Every pattern typing rule (recall Figure 2) examines only structural subformulas of A, except when $A = P$. But any P defined by pure rules $P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+$ in fact has *no* strict subformulas, since the Δ_i such that $\Delta_i \Vdash \langle\Psi\rangle\, A_i^+$ can contain only atomic formulas X^+. $\square$

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 [1992] proof of cut-elimination for the fragment of definitional reflection with $\to$-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, as in Girard [2001], we could adopt a coinductive reading of the rules in Figure 1, in which case identity is always admissible, and cut-elimination is a partial operation that attempts to build a cut-free proof bottom-up. Even under a coinductive reading, we conjecture that cut-elimination is total assuming a positivity restriction for rules, in the sense of Mendler [1987].

3 Logical Properties of $\Rightarrow$ and $\curlywedge$.

3.1 Shock therapy

In §6.2 of "Locus Solum", Girard [2001] considers several "shocking equalities"—counterintuitive properties of the universal and existential quantifiers that emerge when they are given non-standard polarities. For example, positive $\forall$ commutes under $\oplus$, while negative $\exists$ commutes over $\parr$. In our setting, $\Rightarrow$ behaves almost like a positive universal quantifier, and $\curlywedge$ almost like a negative existential.[2] And indeed, we can reproduce analogues of Girard's commutations.

Definition 4. *For two positive contextual formulas C_1^+ and C_2^+, we say that $C_1^+ \lesssim C_2^+$ if $\cdot; C_1^+ \vdash C_2^+$. For negative C_1^- and C_2^-, we say $C_1^- \lesssim C_2^-$ if $C_1^- \vdash C_2^-$. We write $C_1 \approx C_2$ when both $C_1 \lesssim C_2$ and $C_2 \lesssim C_1$. 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. $(R \curlywedge A^-) \& (R \curlywedge B^-) \approx R \curlywedge (A^- \& B^-)$

Proof. Immediate—indeed, in each case, both sides have an isomorphic set of (value/spine) 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. $(\exists x.A) \parr (\exists x.B) \approx \exists x.(A \parr B)$

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 $\text{И}x.A$ of nominal logic [Pitts, 2003], as well as the self-dual ∇ connective of Miller and Tiu [2003]. $\text{И}x.A$ can be interpreted as asserting either that A holds for some fresh name, or for *all* fresh names—with both interpretations being equivalent.

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	::=	$\cdot \mid \Phi, x : \tau$

$$\boxed{\Phi \vdash_\Sigma M : \tau}$$

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

Figure 3: Simply-typed LF

3.2 Embedding of Simply-Typed LF

The canonical forms of simply-typed LF (STLF) are summarized in Figure 3; see Watkins et al. [2002] for an introduction to canonical-forms presentations of logical frameworks. In this section, we show that the STLF terms exist as closed patterns, and therefore as values, in our type theory. This shows that our type theory can represent any logical system that has been represented in STLF.

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 LF base types with our defined atoms P):

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

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

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

Theorem 1 (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 derivations of* $\cdot \Vdash \langle \Psi_\Sigma, \Psi_\Phi \rangle\, A^+$.

Proof. Map $\lambda\, x.\, M$ to the pattern rule for $\Rightarrow$, and map $x\,(M_1, \ldots, M_n)$ to the pattern rule for P. □

This theorem permits us to inherit en masse the adequacy of all systems that have been represented in STLF. For example the above signature Ψ_λ adequately represents the informal syntax of untyped λ-terms because Ψ_λ is the image of the usual LF encoding of this syntax. To complete the discussion of adequacy, we should also check that LF substitution is faithfully modelled in our calculus; to do so, we must consider substitution for rule variables.

3.3 Rule Substitution

As discussed above, there is no reason to expect the rule context Ψ to satisfy substitution in general, but we can prove a generic substitution theorem that covers the STLF fragment. We abuse notation and write L to double both for rules $\mathsf{r}(\tau)$ and types $\mathsf{p}(\tau)$ in the image of the LF encoding.

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

Value pattern	p	::=	$x \mid () \mid (p_1, p_2) \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p \mid u\, p_1 \ldots p_n \mid \lambda\, u.\, p \mid \mathsf{box}\, p$
Spine pattern	k	::=	$\epsilon \mid p\,; k \mid \mathsf{fst};\ k \mid \mathsf{snd}; k \mid \mathsf{out};\ k \mid \mathsf{unpack};\ u.k \mid \mathsf{undia};\ k$
Value	v	::=	$p\,[\sigma]$
Value match	m	::=	$\epsilon \mid \mathsf{match}^{+}(\phi^{+}) \mid \boxed{\epsilon} \mid \boxed{m_2 \circ m_1}$
			where $\phi^{+} ::= \{p_1 \mapsto e_1 \mid \cdots \mid p_n \mapsto e_n\}$
Spine	s	::=	$k[\sigma]; m \mid \boxed{m \circ s}$
Spine match	n	::=	$x \mid \mathsf{match}^{-}(\phi^{-}) \mid \boxed{x} \mid \boxed{\mathsf{fix}(x.n)}$
			where $\phi^{-} ::= \{k_1 \mapsto e_1 \mid \cdots \mid k_n \mapsto e_n\}$
Expression	e	::=	$v \mid s \bullet x \mid \boxed{\mid s \bullet n \mid m \bullet v \mid m \circ e}$
Substitution	σ	::=	$\cdot \mid \sigma, n/x \mid \boxed{\mathsf{id}} \mid \boxed{\sigma_1, \sigma_2}$

Figure 4: Proof Terms

Theorem 2 (Rule Substitution)**.** *For arbitrary Ψ and A^+, for any rule L in the LF fragment, if $\cdot \Vdash \langle \Psi, L \rangle\ A^+$ and $\cdot \Vdash \langle \Psi \rangle\ L$ then $\cdot \Vdash \langle \Psi \rangle\ A^+$.*

Proof. The proof is an adaptation of the standard hereditary substitution algorithm for LF [Watkins et al., 2002] to our pattern syntax. □

In a constructive type theory such as Agda, this theorem is witnessed by a generic function implementing substitution. This function can be applied, for example, to promote a representational arrow $L \Rightarrow A^+$ to a computational arrow $L \rightarrow \uparrow A^+$. Weakening, contraction, exchange, and identity can be proved similarly, and consequently LF-style representations enjoy all the usual structural properties. Representations using more general techniques may enjoy these properties as well, but then a programmer wishing to use a structural property must provide explicit justification that it holds.

It is possible to generalize the above theorem to patterns that bind variables:

Theorem 3 (Rule Substitution With Pattern Variables)**.** *For arbitrary Ψ and A^+, for any rule L in the LF fragment, if $\Delta \Vdash \langle \Psi, L \rangle\ A^+$ and $\Delta_0 \Vdash \langle \Psi \rangle\ L$ then there exists a Δ' such that $\Delta' \Vdash \langle \Psi \rangle\ A^+$. Moreover, for all $X^+ \in \Delta'$, $X^+ \in \Delta_0, \Delta$, and for all $\langle \Psi' \rangle\ A^- \in \Delta'$, there exists a Ψ such that $\langle \Psi \rangle\ A^- \in \Delta_0, \Delta$.*

The conclusion of this theorem is fairly weak: It is always possible to substitute one pattern into another. However, the contexts Ψ' in the types of pattern variables in Δ' may be different from the contexts Ψ in Δ_1, Δ_2 (because the substituted rule L is removed from contexts in Δ, and because contexts in Δ_0 are weakened as the pattern is moved under binders). Thus, it is not necessarily the case that $\Delta_0, \Delta \vdash \Delta'$, so the pattern constructed by the above theorem cannot necessarily be used to form a value in the context Δ_0, Δ. However, in certain circumstances it is possible to produce such substitutions, as we show in an example below.

4 Programming with Definitional Variation

4.1 Proof Terms

In Figure 4, we present a proof term assignment to the focused sequent calculus described above. There is one proof term for each rule in the calculus. For example, the pattern rule for $R \Rightarrow A$ is represented by a binding form $\lambda\, u.\, p$, where the variable represents the new pattern constructor. Additionally, we internalize the cut and identity principles: $s \bullet n$ and $v \bullet m$ witness reduction; $m \circ e$, $m \circ s$, and $m_2 \circ m_1$ witness

$\boxed{e \rightsquigarrow e'}$

$$\frac{\phi^+(p)\ \mathsf{defined}}{\mathsf{match}^+(\phi^+) \bullet p\,[\sigma] \rightsquigarrow \phi^+(p)\,[\sigma]}\ \mathrm{pr} \qquad \frac{}{(m_2 \circ m_1) \bullet v \rightsquigarrow m_2 \circ (m_1 \bullet v)}\ \mathrm{mm}$$

$$\frac{}{\epsilon \bullet v \rightsquigarrow v}\ \mathrm{idm}$$

$$\frac{\phi^-(k)\ \mathsf{defined}}{(k[\sigma]; m) \bullet \mathsf{match}^-(\phi^-) \rightsquigarrow m \circ (\phi^-(k)\,[\sigma])}\ \mathrm{nr} \qquad \frac{}{(m \circ s) \bullet n \rightsquigarrow m \circ (s \bullet n)}\ \mathrm{ms}$$

$$\frac{}{s \bullet \mathsf{fix}(x.n) \rightsquigarrow s \bullet n\,[\mathsf{fix}(x.n)/x]}\ \mathrm{fix}$$

$$\frac{e \rightsquigarrow e'}{m \circ e \rightsquigarrow m \circ e'}\ \mathrm{mee} \qquad \frac{}{m \circ v \rightsquigarrow m \bullet v}\ \mathrm{mev}$$

Figure 5: Operational Semantics

composition; and x, ϵ, and id witness identity. For programming convenience (see Section 4.3), we also internalize the admissible substitution concatenation principle (σ_1, σ_2), a general recursion operator $\mathsf{fix}(x.n)$, (negative) recursive types $\nu\, X^-.A^-$, and two new connectives $\Box A^+$ and $\diamond A^-$. The latter act as modal operators on the rule context: a value of $\langle\Psi\rangle\,\Box A^+$ is a value of $\langle\cdot\rangle\ A^+$; to use $\langle\Psi\rangle\,\diamond A^-$, use $\langle\cdot\rangle\ A^-$. (Like $\Rightarrow$ and $\curlywedge$, these modal connectives are equivalent under polarity shifts.) The full typing rules are presented in Figures 6 and 7. 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).

In Figure 5, we adapt the above cut-elimination procedure into a small-step operational semantics on closed expressions. The rules nr and pr correspond to the negative and positive reduction cases; these rules use an auxiliary operation $e\,[\sigma]$ implementing (standard, capture-avoiding) substitution, which we leave as a meta-operation rather than internalize as a proof term. The rules ms and mm handle instances of composition principles (b) and (c), although in a slightly different manner than above: because the operational semantics consider only closed expressions, it suffices to reassociate the expression to reveal a redex. Similarly, the rules mee and mev simply reduce the expression scrutinized by the case to a value, creating a positive cut. Finally the rule fix unrolls the fixed point, and the rule idm reduces a cut against the internalized identity principle. Type safety can then be formulated as usual, and indeed has an exceptionally easy proof since (as for the cut-elimination procedure in Section 2.3) it need not mention any connectives.

Theorem 4 (Type safety)**.**

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

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

4.2 Implementation

Our type theory is, by design, open-ended with respect to the meta-functions ϕ mapping patterns to expressions. The focusing framework ensures that any means of presenting these meta-functions is acceptable,

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

$$\boxed{\Gamma \vdash v :: C^+}$$

$$\frac{\Delta \Vdash p :: C^+ \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash p\,[\sigma] :: C^+}$$

$$\boxed{\Gamma \vdash m : \gamma_0 > \gamma}$$

$$\frac{}{\Gamma \vdash \epsilon : X^- > X^-} \qquad \frac{\forall(\Delta \Vdash p :: C^+) : \quad \Gamma, \Delta \vdash \phi^+(p) : \gamma}{\Gamma \vdash \mathsf{match}^+(\phi^+) : C^+ > \gamma}$$

$$\boxed{\frac{}{\Gamma \vdash \epsilon : C^+ > C^+}} \qquad \boxed{\frac{\Gamma \vdash m_1 : \gamma_0 > \gamma_1 \quad \Gamma \vdash m_2 : \gamma_1 > \gamma}{\Gamma \vdash m_2 \circ m_1 : \gamma_0 > \gamma}}$$

$$\boxed{\Gamma \vdash s :: C^- > \gamma}$$

$$\frac{\Delta \Vdash k :: C^- > \gamma_0 \quad \Gamma \vdash \sigma : \Delta \quad \Gamma \vdash m : \gamma_0 > \gamma}{\Gamma \vdash k[\sigma]; m :: C^- > \gamma} \qquad \boxed{\frac{\Gamma \vdash s :: C^- > \gamma_0 \quad \Gamma \vdash m : \gamma_0 > \gamma}{\Gamma \vdash m \circ s :: C^- > \gamma}}$$

$$\boxed{\Gamma \vdash n : \alpha}$$

$$\frac{x : X^+ \in \Gamma}{\Gamma \vdash x : X^+} \qquad \frac{\forall(\Delta \Vdash k :: C^- > \gamma) : \quad \Gamma, \Delta \vdash \phi^-(k) : \gamma}{\Gamma \vdash \mathsf{match}^-(\phi^-) : C^-} \qquad \boxed{\frac{x : C^- \in \Gamma}{\Gamma \vdash x : C^-}} \qquad \boxed{\frac{\Gamma, x : C^- \vdash n : C^-}{\Gamma \vdash \mathsf{fix}(x.n) : C^-}}$$

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

$$\frac{\Gamma \vdash v :: C^+}{\Gamma \vdash v : C^+} \qquad \frac{x : C^- \in \Gamma \quad \Gamma \vdash s :: C^- > \gamma}{\Gamma \vdash s \bullet x : \gamma}$$

$$\boxed{\frac{\Gamma \vdash n : C^- \quad \Gamma \vdash s :: C^- > \gamma}{\Gamma \vdash s \bullet n : \gamma} \qquad \frac{\Gamma \vdash v :: C^+ \quad \Gamma \vdash m : C^+ > \gamma}{\Gamma \vdash m \bullet v : \gamma} \qquad \frac{\Gamma \vdash e : \gamma_0 \quad \Gamma \vdash m : \gamma_0 > \gamma}{\Gamma \vdash m \circ e : \gamma}}$$

$$\boxed{\Gamma \vdash \sigma : \Delta}$$

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash \sigma : \Delta \quad \Gamma \vdash n : C^-}{\Gamma \vdash \sigma, n/x : \Delta, x : C^-} \qquad \boxed{\frac{\Delta \subseteq \Gamma}{\Gamma \vdash \mathsf{id} : \Delta}} \qquad \boxed{\frac{\Gamma \vdash \sigma_1 : \Delta_1 \quad \Gamma \vdash \sigma_2 : \Delta_2}{\Gamma \vdash \sigma_1, \sigma_2 : \Delta_1, \Delta_2}}$$

$\boxed{\boxed{\text{identity principles}}}$ $\boxed{\boxed{\text{cut principles}}}$ $\boxed{\boxed{\text{convenient principles}}}$

Figure 6: Focusing rules with proof terms

Pos. formula	$A^+ ::= X^+ \mid {\downarrow}A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+ \mid P \mid R \Rightarrow A^+ \mid \Box A^+$
Rule	$R ::= P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+$
Rule Context	$\Psi ::= \cdot \mid \Psi, u : R$
Neg. formula	$A^- ::= X^- \mid {\uparrow}A^+ \mid A^+ \to B^- \mid \top \mid A^- \& B^- \mid \nu X^-.A^- \mid R \curlywedge B^- \mid \diamond A^-$
CPF	$C^+ ::= \langle \Psi \rangle\, A^+$
CNF	$C^- ::= \langle \Psi \rangle\, A^-$

$$\boxed{\Delta \Vdash p :: \langle \Psi \rangle\, A^+}$$

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

$$\frac{}{\cdot \Vdash () :: \langle \Psi \rangle\, 1} \qquad \frac{\Delta_1 \Vdash p_1 :: \langle \Psi \rangle\, A^+ \quad \Delta_2 \Vdash p_2 :: \langle \Psi \rangle\, B^+}{\Delta_1, \Delta_2 \Vdash (p_1, p_2) :: \langle \Psi \rangle\, A^+ \otimes B^+}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Vdash p :: \langle \Psi \rangle\, A^+}{\Delta \Vdash \mathsf{inl}\, p :: \langle \Psi \rangle\, A^+ \oplus B^+} \qquad \frac{\Delta \Vdash p :: \langle \Psi \rangle\, B^+}{\Delta \Vdash \mathsf{inr}\, p :: \langle \Psi \rangle\, A^+ \oplus B^+}$$

$$\frac{\begin{array}{c} u : P \Leftarrow A_1^+ \Leftarrow \ldots \Leftarrow A_n^+ \in (\Sigma, \Psi) \\ \Delta_1 \Vdash p_1 :: \langle \Psi \rangle\, A_1^+ \quad \ldots \quad \Delta_n \Vdash p_n :: \langle \Psi \rangle\, A_n^+ \end{array}}{\Delta_1, \ldots, \Delta_n \Vdash u\, p_1 \ldots p_n :: \langle \Psi \rangle\, P} \qquad \frac{\Delta \Vdash p :: \langle \Psi, u : R \rangle\, B^+}{\Delta \Vdash \lambda\, u.\, p :: \langle \Psi \rangle\, R \Rightarrow B^+} \qquad \frac{\Delta \Vdash p :: \langle \cdot \rangle\, A^+}{\Delta \Vdash \mathsf{box}\, p :: \langle \Psi \rangle\, \Box A^+}$$

$$\boxed{\Delta \Vdash k :: \langle \Psi \rangle\, A^- > \gamma}$$

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

$$\frac{\Delta_1 \Vdash p :: \langle \Psi \rangle\, A^+ \quad \Delta_2 \Vdash k :: \langle \Psi \rangle\, B^- > \gamma}{\Delta_1, \Delta_2 \Vdash p\, ; k :: \langle \Psi \rangle\, A^+ \to B^- > \gamma}$$

$$\text{(no rule for } \top \text{)} \qquad \frac{\Delta \Vdash k :: \langle \Psi \rangle\, A^- > \gamma}{\Delta \Vdash \mathsf{fst};\ k :: \langle \Psi \rangle\, A^- \& B^- > \gamma} \qquad \frac{\Delta \Vdash k :: \langle \Psi \rangle\, B^- > \gamma}{\Delta \Vdash \mathsf{snd}; k :: \langle \Psi \rangle\, A^- \& B^- > \gamma}$$

$$\frac{\Delta \Vdash k :: \langle \Psi \rangle\, [\nu X^-.A^- / X^-]A^- > \gamma}{\Delta \Vdash \mathsf{out};\ k :: \langle \Psi \rangle\, \nu X^-.A^- > \gamma}$$

$$\frac{\Delta \Vdash k :: \langle \Psi, u : R \rangle\, B^- > \gamma}{\Delta \Vdash \mathsf{unpack};\ u.k :: \langle \Psi \rangle\, R \curlywedge B^- > \gamma} \qquad \frac{\Delta \Vdash k :: \langle \cdot \rangle\, A^- > \gamma}{\Delta \Vdash \mathsf{undia};\ k :: \langle \Psi \rangle\, \diamond A^- > \gamma}$$

Figure 7: Value and spine patterns

provided only that they define total functions mapping patterns to expressions. One could, for example, extend our theory with a formalism for meta-functions that are defined by primitive recursion over patterns. However, experience (e.g., [Harper and Morrisett, 1995]) has shown that such restricted formalisms are very difficult to use in practice, since they amount to a commitment to a particular form of termination proof. Instead we prefer to harness the expressive power of existing tools to prove meta-functions total.

Thus, an implementation of our type theory must include an embedding of its syntax into the language of some (perhaps more than one) proof assistant in which we are to carry out proofs of totality. We may use any means of representing patterns and expressions, such as de Bruijn indices or locally nameless form, and any method for proving such functions total. We are also free to make use of syntactic support for translating concrete named forms into a name-free form suitable for that environment. The focusing framework ensures that all that is required of a proof assistant is a means of computing $\phi^+(p)$ and $\phi^-(k)$ for given meta-functions ϕ^+ and ϕ^- and arguments p and k, respectively.

At present, we have built simple embeddings of our language in Agda and Coq, both using de Bruijn index representations.[3] In the remainder of this section, we use a convenient surface syntax to present the examples; the interested reader may refer to the companion Agda code for a more precise account. It may also be possible to embed our language into Twelf, using higher-order abstract syntax, and relying on the Twelf totality and coverage checkers to check that meta-functions, presented as relations between patterns and expressions, are total functions. The logic programming interpretation of types in Twelf would then provide the means of computing $\phi^+(p)$ and $\phi^-(k)$, with termination being ensured by the totality checker.

4.3 Examples

4.3.1 Induction over First-Order Data

In this section, we instantiate our type theory with a global definition signature for natural numbers $\Sigma_{\mathbb{N}} = \langle \mathsf{Z} : \mathsf{nat}\,;\ \mathsf{S} : \mathsf{nat} \Leftarrow \mathsf{nat} \rangle$. As a most basic programming example, we define a function *add1*, which simply wraps the successor pattern constructor S as a function on natural numbers:

$$\begin{array}{l} \cdot \vdash \mathit{add1} : \langle\rangle\, \mathsf{nat} \to {\uparrow}\mathsf{nat} \\ \mathit{add1} = \mathsf{match}^-(p\,;\epsilon \mapsto (\mathsf{S}\ p)\,[\mathsf{id}]) \end{array}$$

The body of this spine match is a meta-level function (i.e., an Agda function in our Agda implementation) that maps spine patterns to expressions. In particular, for every spine pattern $\Delta \Vdash k :: \langle\rangle\,\mathsf{nat} \to {\uparrow}\mathsf{nat} > \gamma$, it must give an expression $\Delta \vdash e : \gamma$. In this case, all spine patterns for $\langle\rangle\,\mathsf{nat} \to {\uparrow}\mathsf{nat}$ have the form $p\,;\epsilon$, where $\Delta \Vdash p :: \langle\rangle\,\mathsf{nat}$ and $\gamma = \langle\rangle\,\mathsf{nat}$. In $\Sigma_{\mathbb{N}}$, every such p is a numeral, so extensionally, we wish to define the following meta-function:

$$\begin{array}{rcl} \mathsf{Z}\,;\epsilon & \mapsto & (\mathsf{S}\ \mathsf{Z})\,[\mathsf{id}] \\ \mathsf{S}\ \mathsf{Z}\,;\epsilon & \mapsto & (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))\,[\mathsf{id}] \\ \mathsf{S}\ (\mathsf{S}\ \mathsf{Z})\,;\epsilon & \mapsto & (\mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z})))\,[\mathsf{id}] \\ \vdots & \mapsto & \vdots \end{array}$$

In each case, this function adds a successor to the pattern, and places this new pattern under the identity substitution (we abbreviate $\cdot, \mathsf{id}$ by id) to form a value and therefore an expression. However, we cannot

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

list a case for each numeral explicitly, so in the above code $p\,;\epsilon \mapsto (\mathsf{S}\ p)\,[\mathsf{id}]$ we use a *metavariable* p to treat all such patterns uniformly. The metavariable is a variable, ranging over patterns, in the meta-language (e.g., an Agda variable in our Agda implementation), permitting a finite representation of the infinite set of ordered pairs in the extension of the function. Note that, because numerals bind no variables in Δ, we could equivalently have used the empty substitution $\cdot$ in place of the identity substitution id; however, the type-correctness of this modification would require the inductive proof that $\Delta \Vdash p :: \langle\rangle\,\mathsf{nat}$ implies $\Delta = \cdot$, whereas the identity-substitution version is type-correct for any Δ.

Next, we define addition as follows:

$$\begin{array}{l} \cdot \vdash \mathit{plus} : \langle\rangle\,\mathsf{nat} \to \mathsf{nat} \to \uparrow\!\mathsf{nat} \\ \mathit{plus} = \mathsf{match}^-(p_1\,;p_2\,;\epsilon \mapsto \mathit{plus}^*\ p_1\ p_2) \end{array}$$

That is, every spine pattern decomposing $\langle\rangle\,\mathsf{nat} \to \mathsf{nat} \to\uparrow\!\mathsf{nat}$ is composed of two value patterns, which are matched by the metavariables p_1 and p_2, and then passed to an auxiliary meta-function plus^*:

$$\begin{array}{llll} \multicolumn{4}{l}{\forall(\Delta \Vdash p_1 :: \langle\rangle\,\mathsf{nat},\ \Delta_2 \Vdash p_2 :: \langle\rangle\,\mathsf{nat}) :\ \ (\Delta_1, \Delta_2 \vdash \mathit{plus}^*\ p_1\ p_2 : \mathsf{nat})} \\ \mathit{plus}^*\ \ \mathsf{Z} & p_2 & \mapsto & p_2\,[\mathsf{id}] \\ \mathit{plus}^*\ \ (\mathsf{S}\ p_1) & p_2 & \mapsto & \mathsf{match}^+(n \mapsto ((n\,;\epsilon)[\mathsf{id}];\epsilon) \bullet \mathit{add1}) \circ (\mathit{plus}^*\ p_1\ p_2) \end{array}$$

This auxiliary function maps two patterns to an expression that computes their sum, using meta-level case-analysis and induction on the first pattern. The result of the second branch can be made much more readable by employing a bit of syntactic sugar: we write $\mathsf{case}\ e\ \mathsf{of}\ \phi^+$ for $\mathsf{match}^+(\phi^+) \circ e$, write $s \bullet n$ as $n\ s$, and elide both the ϵ at the end of spine patterns and the identity case ϵ following a spine pattern and substitution. Then this branch is written as follows:

$$\mathit{plus}^*\ (\mathsf{S}\ p_1)\,p_2 \quad\mapsto\quad \mathsf{case}\,(\mathit{plus}^*\ p_1\ p_2)\,\mathsf{of}\ n \mapsto \mathit{add1}\ (n[\mathsf{id}])$$

The case explicitly sequences the evaluation of the expression produced by the inductive call ($\mathit{plus}^*\ p_1\ p_2$) down to a value, which is matched by the metavariable n. This sequencing is necessary because spines can be composed only of values, not arbitrary expressions. As an example invocation, $\mathit{plus}^*\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{Z}))\ (\mathsf{S}\ \mathsf{Z})$ builds the expression

$$\mathsf{case}\ (\mathsf{case}\,(\mathsf{S}\ \mathsf{Z})\,[\mathsf{id}]\,\mathsf{of}\ n \mapsto \mathit{add1}\ (n[\mathsf{id}]))\ \mathsf{of}\ n \mapsto \mathit{add1}\ (n[\mathsf{id}])$$

which computes as expected under the above operational semantics.

4.3.2 Induction Over Data With Binding

Next, we illustrate recursion over data with variable binding. As a simple first example, we count the variables (leaves) of an untyped λ-term. We work in the signature

$$\Sigma_\lambda = \Sigma_{\mathbb{N}}\ ,\ \mathsf{lam} : \mathsf{exp} \Leftarrow (\mathsf{exp} \Rightarrow \mathsf{exp})\ ,\ \mathsf{app} : \mathsf{exp} \Leftarrow \mathsf{exp} \Leftarrow \mathsf{exp}$$

In general, a computational function such as *add1* or *plus* does not necessarily remain well-typed when the signature Σ is extended (i.e., signatures may not necessarily be weakened): the extension could create new cases that the function does not handle. However, in this case the functions *add1* and *plus* remain well-typed, since the constants lam and app do not change the definition of nat.

Closed Terms First, we implement a function *cv* that counts the variables of a closed λ-term (generalizations to open terms are considered below):

$$\begin{array}{l} \cdot \vdash \mathit{cv} : \langle\rangle\, \mathsf{exp} \rightarrow \uparrow \mathsf{nat} \\ \mathit{cv} = \mathsf{match}^{-}(p\,;\epsilon \mapsto \mathit{cv}^{*}\ p) \\ \\ \forall(\Delta \Vdash p :: \langle\rangle\, \mathsf{exp}) : \quad (\Delta \vdash \mathit{cv}^{*}\ p : \langle\rangle\, \mathsf{nat}) \end{array}$$

The body of *cv* is a meta-function cv^*, which must map patterns representing closed λ-terms to expressions of type $\langle\rangle\,\mathsf{nat}$. However, when we attempt to define a meta-function with this type, we will run into a problem in the lam-case:

$$\mathit{cv}^{*}\ (\mathsf{lam}(\lambda\, u.\, p)) \quad \mapsto \quad ???$$

We would like to make a recursive call to count the variables in the body of the function, which is a pattern $\Delta \Vdash p :: \langle u : \mathsf{exp}\rangle\, \mathsf{exp}$. However, the meta-function cv^* is stated only for closed λ-terms, represented by patterns $\Delta \Vdash p' :: \langle\rangle\, \mathsf{exp}$.

The solution is to generalize the type of cv^* so that it is defined simultaneously for all rule contexts Ψ_{v} of the form $u_1 : \mathsf{exp}, \ldots, u_n : \mathsf{exp}$. (In the setting of Twelf, the analogous technique is proving a metatheorem in a regular world [Pfenning and Schürmann, 1999].)

$$\begin{array}{llll} \multicolumn{4}{l}{\forall(\Psi_{\mathrm{v}}\,,\, \Delta \Vdash p :: \langle\Psi_{\mathrm{v}}\rangle\, \mathsf{exp}) : \quad \Delta \vdash (\mathit{cv}^{*}\ \Psi_{\mathrm{v}}\ p) : \langle\rangle\, \mathsf{nat}} \\ \mathit{cv}^{*} & \Psi & u \text{ where } u \in \Psi & \mapsto \quad (\mathsf{S\ Z})\,[\mathsf{id}] \\ \mathit{cv}^{*} & \Psi & \mathsf{lam}\,(\lambda\, u.\, p) & \mapsto \quad \mathit{cv}^{*}\ (\Psi, u : \mathsf{exp})\ p \\ \mathit{cv}^{*} & \Psi & \mathsf{app}\ p_1\ p_2 & \mapsto \quad \mathsf{case}\,(\mathit{cv}^{*}\ \Psi\ p_1)\,\mathsf{of}\ n_1 \mapsto \mathsf{case}\,(\mathit{cv}^{*}\ \Psi\ p_2)\,\mathsf{of}\ n_2 \mapsto \mathit{plus}\ (n_1\,;n_2)[\mathsf{id}] \end{array}$$

Here, we write the context Ψ as an explicit argument to the meta-function; however, because Ψ appears in the type of p, it should be possible to infer this argument—indeed, our Agda implementation of this function treats Ψ as an implicit argument. Because result of cv^* is an expression of type nat in the empty rule context, we may call to the previously-defined addition function, which is defined only for closed nats, *plus* in the app case.

We complete *cv* by calling cv^* in the empty rule context: $\mathit{cv} = \mathsf{match}^{-}(p\,;\epsilon \mapsto \mathit{cv}^{*}\ \cdot\ p)$.

Terms in a Fixed Context The above function *cv* of type $\langle\rangle\,\mathsf{exp} \rightarrow \uparrow\mathsf{nat}$ can only be applied to a closed λ-term. As a first generalization, we write an analogous function that can be applied to any λ-term in an arbitrary, but fixed, context Ψ_{v}:

$$\cdot \vdash \mathit{cv}' : \langle\Psi_{\mathrm{v}}\rangle\, \mathsf{exp} \rightarrow \uparrow\mathsf{nat}$$

The meta-function cv^* above was stated generally for any such Ψ_{v}, so we should be able to reuse it to implement cv'. However, there is a slight difference between cv^* and the meta-function required to implement cv'. The type $\langle\Psi_{\mathrm{v}}\rangle\,\mathsf{exp} \rightarrow \uparrow\mathsf{nat}$ classifies functions that consume λ-terms in Ψ_{v} and produce numbers in Ψ_{v}. Formally, to implement a function of type $\langle\Psi_{\mathrm{v}}\rangle\,\mathsf{exp} \rightarrow \uparrow\mathsf{nat}$ it suffices to give a meta-function $\forall(\Delta \Vdash p :: \langle\Psi_{\mathrm{v}}\rangle\,\mathsf{exp}) : \Delta \vdash e : \langle\Psi_{\mathrm{v}}\rangle\,\mathsf{nat}$. In contrast, $(\mathit{cv}^*\ \Psi_{\mathrm{v}})$ consumes λ-terms in Ψ_{v} but produces *closed* numbers.

To use cv^* to implement cv', we have two choices. First, we may explicitly coerce the result of cv^* from $\langle\rangle\,\mathsf{nat}$ to $\langle\Psi_{\mathrm{v}}\rangle\,\mathsf{nat}$. Such a coercion function is definable because weakening with exp variables is admissible for nat values—essentially because the assumptions $x : \mathsf{exp}$ in Ψ_{v} do not alter the definition of nat. Second, we may refine the type of cv' to capture the invariant that the result is closed. To do so, we use a connective $\diamond A^-$, which classifies an A^- in the empty rule context (see Figure 7 for its spine rule). Then we can implement *cv′* as follows:

$$\begin{array}{l} \cdot \vdash \mathit{cv}' : \langle \Psi_{\mathrm{v}} \rangle\, \mathsf{exp} \to \diamond{\uparrow}\mathsf{nat} \\ \mathit{cv}' = \mathsf{match}^-(p\,; \mathsf{undia};\, \epsilon) \mapsto \mathit{cv}^*\ \Psi_{\mathrm{v}}\ p \end{array}$$

Unlike *cv*, the function *cv′* can be applied to any λ-term constructed from variables in Ψ_{v}.

Regular Worlds Finally, we consider representing Twelf's regular worlds in our type system. Regular worlds are a kind of context polymorphism: rather than proving a theorem for a fixed context, you prove a theorem for all contexts of a particular shape. Consequently, it is permissible to appeal to the theorem from any context in the world. To recur under binders, it is often necessary to generalize a theorem statement about closed terms to a regular world containing variables. Our meta-function *cv** is an example of this phenomenon: to recur in the lam case, we wrote a function that can be called from any context Ψ_{v} in the regular world exp^*, which contains contexts of the form $u_1 : \mathsf{exp}, \ldots, u_n : \mathsf{exp}$. We now show how to internalize this generality as a type.

We would like a type $\forall_{\mathsf{exp}^*}(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat})$ with the property that we can call this function on any λ-term in any context Ψ_{v} in exp^*. Such a type is equivalent to the conjunction of the following infinite list of contextual types:

$$\begin{array}{l} \langle\rangle\, \mathsf{exp} \to \diamond{\uparrow}\mathsf{nat} \\ \langle u_1 : \mathsf{exp}\rangle\, \mathsf{exp} \to \diamond{\uparrow}\mathsf{nat} \\ \langle u_1 : \mathsf{exp}, u_2 : \mathsf{exp}\rangle\, \mathsf{exp} \to \diamond{\uparrow}\mathsf{nat} \\ \qquad\vdots \end{array}$$

To give a finite representation of this list, we first internalize each fixed context Ψ_{v} using the $\curlywedge$ connective:

$$\begin{array}{l} \langle\rangle\, (\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}) \\ \langle\rangle\, \mathsf{exp} \curlywedge (\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}) \\ \langle\rangle\, \mathsf{exp} \curlywedge (\mathsf{exp} \curlywedge (\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat})) \\ \qquad\vdots \end{array}$$

Because of this internalization, all of these types are in the same (empty) rule context, which enables us to conjoin them. Thus, we could prove them all at once by proving the infinitary conjunction

$$\langle\rangle\ \&_{n=0}^{\infty}(\mathsf{exp}^n \curlywedge (\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}))$$

where $\mathsf{exp}^n \curlywedge A^-$ iterates $(\mathsf{exp} \curlywedge \_)$ n times. While our type theory does not have infinite conjunctions as a primitive notion, we can define them using a (negative) recursive type. In this case, we use the following type:

$$\nu\, X^-.(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}) \& (\mathsf{exp} \curlywedge X^-)$$

(See Figure 7 for the typing rule for $\nu\, X^-.A^-$.) The type $\nu\, X^-.A^- \& (\mathsf{exp} \curlywedge X^-)$ is an A^- under a variable number of new constants of type exp. To extract a function $\langle (u : \mathsf{exp})^n \rangle\, \mathsf{exp} \to {\uparrow}\mathsf{nat}$, we use the spine pattern $((\mathsf{out};\, \mathsf{snd};\, \mathsf{unpack})^n; \mathsf{out};\, \mathsf{fst})$ from this type. We abbreviate this recursive type using the regular worlds syntax $\forall_{\mathsf{exp}^*}(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat})$.

And indeed, *cv** suffices to implement the type $\cdot \vdash \mathit{cv}'' : \langle\rangle\, \forall_{\mathsf{exp}^*}(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat})$. The implementation uses an auxiliary inversion lemma:

$$\forall(\Delta \Vdash k :: \langle \Psi_{\mathrm{v}} \rangle\, \forall_{\mathsf{exp}^*}(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}) > \gamma) : \quad \exists \Psi'_{\mathrm{v}}.\ \cdot \Vdash \mathit{invert}\ k :: \langle \Psi'_{\mathrm{v}} \rangle\, \mathsf{exp} \text{ and } \gamma = \langle\rangle\ \mathsf{nat}$$

This lemma transforms any spine pattern of this type into an exp pattern in some context Ψ'_{v}, and is defined defined by a simple induction over the spine patterns for the recursive type. Then cv'' is defined as follows:

$$\begin{array}{l} \cdot \vdash \mathit{cv}'' : \langle\rangle\, \forall_{\mathsf{exp}^*}(\mathsf{exp} \to \diamond{\uparrow}\mathsf{nat}) \\ \mathit{cv}'' = \mathsf{match}^-(k \mapsto \mathit{cv}^*\ \Psi'_{\mathrm{v}}\ (\mathit{invert}\ k)) \end{array}$$

Thus, our type theory provides an analogue of Twelf's regular worlds using $\curlywedge$, $\&$, and ν.

4.3.3 Using Substitution

In Theorem 2, we proved a substitution theorem for LF-like rules in closed patterns. The proof of this theorem defines a meta-function from patterns to patterns:

$$\forall(\cdot \Vdash p_0 :: \langle\Psi\rangle\, L \text{ and } \cdot \Vdash p :: \langle\Psi, u : L\rangle\, A^+) : \quad \cdot \Vdash [p_0/u]p :: \langle\Psi\rangle\, A^+$$

We can exploit this theorem to give a simple definition of β-reduction on untyped λ-terms: there is no need to define substitution explicitly for this individual object language. We define a function *red* that reduces a term one step, if such a reduction is possible; the body of this function is the following meta-function:

$$\forall(\Psi_{\mathrm{v}} \text{ and } \Delta \Vdash p :: \langle\Psi_{\mathrm{v}}\rangle\, \mathsf{exp}) : \quad \Delta \vdash \mathit{red}^*\ p : \langle\Psi_{\mathrm{v}}\rangle\, \mathsf{exp} \oplus 1$$

$$\begin{array}{lll}
\mathit{red}^*\ \ u \text{ where } u \in \Psi & \mapsto & \mathsf{inr}\,()\,[\cdot] \\
\mathit{red}^*\ \ \mathsf{lam}\,(\lambda\, u.\, p) & \mapsto & \mathsf{case}\,\mathit{red}^*\ p\ \mathsf{of} \\
 & & \quad \mathsf{inr}\,() \mapsto \mathsf{inr}\,()\,[\cdot] \\
 & & \quad \mathsf{inl}\,p' \mapsto \mathsf{inl}\,(\mathsf{lam}\ \lambda\, u.\, p')\,[\mathsf{id}] \\
\mathit{red}^*\ \ \mathsf{app}\,(\mathsf{lam}\,(\lambda\, u.\, p))\ p_2 & \mapsto & \mathsf{inl}\,([p_2/u]p)\,[\cdot] \\
\mathit{red}^*\ \ \mathsf{app}\ p_1\ p_2 & \mapsto & \mathsf{case}\,\mathit{red}^*\ p_1\ \mathsf{of} \\
 & & \quad \mathsf{inr}\,() \mapsto \mathsf{case}\,\mathit{red}^*\ p_2\ \mathsf{of} \\
 & & \qquad\qquad \mathsf{inr}\,() \mapsto \mathsf{inr}\,()\,[\cdot] \\
 & & \qquad\qquad \mathsf{inl}\,p'_2 \mapsto \mathsf{inl}\,(\mathsf{app}\ p_1\ p'_2)\,[\mathsf{id}] \\
 & & \quad \mathsf{inl}\,p'_1 \mapsto \mathsf{inl}\,(\mathsf{app}\ p'_1\ p_2)\,[\mathsf{id}]
\end{array}$$

In this definition, we leave the context argument Ψ implicit—it can be inferred from the type of the pattern argument. The appeal to substitution in the β-redex case is type-correct because any pattern p such that $\Delta \Vdash p :: \langle\Psi\rangle\,\mathsf{exp}$ in fact satisfies $\cdot \Vdash p :: \langle\Psi\rangle\,\mathsf{exp}$—because there are no shifted negative formulae as hypotheses of the rules defining exp. This meta-function implements the recursive type $\cdot \vdash \mathit{red} : \langle\rangle\, \forall_{\mathsf{exp}^*}(\mathsf{exp} \to {\uparrow}(\mathsf{exp} \oplus 1))$, similarly to cv'' above.

4.3.4 Mixing Representational and Computational Functions

Next, to illustrate the combination of representational and computational arrows in a single set of rules, we represent 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 $\mathsf{nat} \to \mathsf{nat} \to {\uparrow}\mathsf{nat}$. The language includes numeric constants, binops, and let-binding, and is represented by Σ_{ari}:

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

For example, the value $(\mathsf{binop}\ (\mathsf{num}\ 4)\ f\ (\mathsf{num}\ 5))\ [\mathit{plus}/f]$ represents an addition instruction. Observe that *plus* remains well-typed in this extended signature.

We can implement an evaluator for closed programs using a fixed point at type $A_{ev} = \langle\rangle\ \mathsf{ari} \to \uparrow\mathsf{nat}$:

$$\begin{array}{l}\cdot \vdash \mathit{eval} : A_{ev}\\ \mathit{eval} = \mathsf{fix}(ev.\mathsf{match}^{-}(p\,;\epsilon \mapsto \mathit{ev}^{*}\ p))\end{array}$$

Then the body of the spine match is defined as follows, where the variable ev is the recursive reference:

$$\begin{array}{lll}\forall(\Delta \Vdash p :: \langle\rangle\ \mathsf{ari}) : & (ev : A_{ev}, \Delta \vdash (\mathit{ev}^{*}\ p) : \langle\rangle\ \mathsf{nat}) \\ \mathit{ev}^{*}\ \ \mathsf{num}\ n & \mapsto & n\,[\mathsf{id}] \\ \mathit{ev}^{*}\ \ \mathsf{binop}\ p_1\ f\ p_2 & \mapsto & ev\ (p_1[\mathsf{id}]; \mathsf{match}^{+}(n_1 \mapsto ev\ (p_2[\mathsf{id}]; \mathsf{match}^{+}(n_2 \mapsto f\ (n_1\,; n_2[\mathsf{id}]))))) \\ \mathit{ev}^{*}\ \ \mathsf{let}\ p_0\ (\lambda\, u.\, p) & \mapsto & ev\ (\{p_0/u\}p[\sigma_{tr}])\end{array}$$

In the binop case, evaluation calls the embedded computational function on the values of the arguments. In the let case, evaluation substitutes the argument (a pattern $\Delta_0 \Vdash p_0 :: \langle\rangle\ \mathsf{ari}$) into the body (a pattern $\Delta \Vdash p :: \langle u : \mathsf{ari}\rangle\ \mathsf{ari}$) and evaluates the result. The explicit fixed point is necessary because the recursive call on the result of substitution is not structural. The pattern substitution operation $\{\_/\_\}\_$ implements the proof of Theorem 3, and produces a pattern $\Delta' \Vdash \{p_0/u\}p :: \langle\rangle\ \mathsf{exp}$, where Δ' is related to Δ_0, Δ as prescribed by that theorem. To complete this case, we must give a substitution $\Delta_0, \Delta \vdash \sigma_{tr} : \Delta'$.

This substitution is constructed by the following reasoning: All rule contexts Ψ arising in patterns $\Delta \Vdash p :: \langle\rangle\ \mathsf{ari}$ have the form $u_1 : \mathsf{ari}, \ldots, u_n : \mathsf{ari}$, and all pattern variable contexts Δ contain only assumptions $x : \langle\Psi\rangle\ \mathsf{nat} \to \mathsf{nat} \to \uparrow\mathsf{nat}$ for such a context Ψ. Theorem 3 shows that for every assumption $f' : \langle\Psi'\rangle\ \mathsf{nat} \to \mathsf{nat} \to \uparrow\mathsf{nat}$ in Δ', there is an assumption $f : \langle\Psi\rangle\ \mathsf{nat} \to \mathsf{nat} \to \uparrow\mathsf{nat}$ in Δ_0, Δ, and Ψ contains only ari assumptions as well. Thus, we can show $\Delta_0, \Delta \vdash \sigma_{tr} : \Delta'$ if we can show how to transport a function f of type $\mathsf{nat} \to \mathsf{nat} \to \uparrow\mathsf{nat}$ from Ψ to Ψ'. Because ari assumptions are irrelevant to nat, we can write a function transporting a nat from Ψ to Ψ' (or vice versa), and then use this function to wrap f. Our Agda code formalizes this reasoning and uses it to construct σ_{tr}. This reasoning is similar to Twelf's technology for *subordination-based world subsumption*; in future work, we plan to consider whether Twelf's technology can be adapted to infer these substitution transformers automatically in certain cases.

4.4 Differences with Agda Implementation

We have manually compiled the above examples to the Agda implementation of our type theory. The Agda versions of these examples differ from the above as follows:

- Variables, both from Γ and from Ψ, are represented as de Bruijn indices.
- Some explicit calls to a weakening lemma for Γ are inserted, in order to weaken some expressions computed by meta-functions.
- The premise $\Delta \subseteq \Gamma$ of the typing rule for the identity substitution id is proved explicitly whenever the identity substitution is used.

- A lemma that $\Delta \Vdash p :: \langle\Psi_{\mathrm{v}}\rangle\,\mathsf{exp}$ implies $\Delta = \cdot$, which justifies the call to the generic substitution routine from *red**, is proved and used.
- The above informal description of the construction of σ_{tr} is made formal.

We leave the formal definition of a surface syntax and elaboration algorithm for meta-functions to future work.

5 Related Work

In Section 1, we highlighted some contrasts between our work and concrete implementations of binding, nominal logic, and HOAS in LF. In summary: The concrete implementations are complementary to the present work, in that the representational arrow $\Rightarrow$ provides a *logical interface* for binding and scope, while leaving the implementation of this interface abstract. Moreover, the computational open-endedness of $\rightarrow$ permits programmers who are fond of a particular implementation of binding to program directly using that implementation, while still ensuring compatibility with other implementations. In this sense, our calculus provides functionality similar to Ott [Sewell et al., 2007], a proof-assistant-independent tool for describing binding structures; however, we provide a proof-assistant-independent account of computation with binding as well. The central difference between our approach and nominal logic is that all names in nominal logic have global scope, whereas the connectives $\Rightarrow$ and $\curlywedge$ intrinsically capture the notion of scope. However, the connectives $\Rightarrow$ and $\curlywedge$ share the some/any coincidence of the self-dual connective И [Pitts, 2003]. The chief difference between the representational function space and HOAS in LF is that, like names in nominal logic, our rule variables are not *a priori* committed to structural properties such as substitution. Instead, representational functions are eliminated by pattern matching, which is similar to the elimination form used to reason *about* LF terms in computational languages such as Twelf [Pfenning and Schürmann, 1999], Delphin [Poswolsky and Schürmann, 2008], and Pientka's recent work [2008] based on contextual modal type theory [Nanevski et al., 2007]. Whereas we generalize LF by allowing computations in rules, all of these approaches treat LF as a pure fragment, though Pientka discusses the potential usefulness of allowing some form of computation within signatures.

The idea of representing variable binding by a function space is very old, going back at least to Church [1940]. However, integrating higher-order representations with computation in a single language has proved difficult. The essential impediment is that one cannot simultaneously represent syntax as an inductive definition inside a type theory and adequately represent binding as the full computational function space of that type theory. There are two reasons for this: First, it is is not permissible to place a type to the left of a computational arrow in its own inductive definition. Second, even if one could, there would be many more computational functions of such a type than the uniform ones that adequately represent binding. This dilemma can be resolved in various ways. LF gives up on treating syntax as an inductive definition inside the type theory. Internally to LF, binding is represented by a "computational" arrow, in the sense that the LF function space has the usual implication rules. However, LF functions have no interesting data to compute with, because object-language syntax is represented by uninterpreted base types. This necessitates a stratified approach, where the terms of LF are externally treated as data in a separate language, such as Twelf, with its own computational arrow (from this external perspective, the LF function space serves a representational role). In this work, we take the alternative approach: we represent object-language syntax by an inductive type, but we do not represent binding by computational functions. Our representational functions, which reason from a fresh constant, are much less powerful than computational functions, which circumscribe the entirety of their domain. Consequently, it is not problematic to put a type to the left of a representational arrow in its inductive definition, as our cut admissibility proof in Section 2 shows. Moreover, as we proved in Section 3, our representational functions adequately represent binding.

Several other single-language resolutions of this dilemma have been studied. Schürmann et al. [2001] describe an approach based on making a type distinction between representational and computational functions. In their language, a modal type $\Box A$ classifies closed terms of type A and is eliminated by primitive recursion; Despeyroux and Leleu [1999] describe a generalization with dependent types. Unlike the LF-based approaches described above, these languages are not syntactically stratified into separate representational and computational parts. Instead, they reuse the same arrow for representational functions (e.g., a function $A \rightarrow B$ where A and B do not include $\Box$, is used for representation) and computational functions (e.g., a function $\Box A \rightarrow B$ can decompose A by primitive recursion). However, unlike the present work, these languages do not permit computational functions such as $\Box A \rightarrow B$ in datatype definitions.

Despeyroux et al. [1995] propose representing object-language syntax as an inductive type (say, exp) in Coq, where a binder is represented using a Coq computational function $\mathsf{var} \rightarrow \mathsf{exp}$. This "weak" higher-order abstract syntax is a valid inductive definition because the domain of computational functions representing binders is a separate type var representing variables. This representation encodes object-language α-equivalence using Coq α-equivalence, but object-language substitution must be programmed explicitly. Care must still be taken to ensure adequacy. Despeyroux et al. [1995] expose the implementation of the type var (e.g., natural numbers), which necessitates a predicate recognizing those functions of type $\mathsf{var} \rightarrow \mathsf{exp}$ which adequately represent object-language terms. Another approach is to leave the type var abstract, though programming with such a representation requires some extra semantically justified axioms about variables [Bucalo et al., 2006]. Ambler et al. [2003] present a variation on weak HOAS in which free variables are represented as projections from an infinite context (a stream of variables); this approach mitigates Isabelle's lack of dependent types, which could be used to characterize terms' contexts more precisely.

Another solution is to give an interface based on computational functions, but implement it using a concrete representation. For example, the Hybrid approach [Ambler et al., 2002, Capretta and Felty, 2007, Momigliano et al., 2007], which has been implemented in Isabelle and Coq, uses an underlying de Bruijn implementation of binding. This circumvents the problems with using computational functions directly in an inductive definition. The challenge is then to provide a useful higher-order interface to this concrete implementation. Providing higher-order constructors is not difficult: for example, a constructor $\mathsf{lam} : (\mathsf{exp} \rightarrow \mathsf{exp}) \rightarrow \mathsf{exp}$ (where exp is the type of the de Bruijn implementation) can be implemented by applying the argument function to an appropriate de Bruijn index. However, to compute or reason using the higher-order interface, it is necessary to restrict attention to those computational functions that adequately represent binders. In Hybrid, this is accomplished by defining a predicate recognizing those computational functions whose application is equivalent to the de Bruijn implementation of substitution—i.e., recognizing substitution functions. While the setting and technical details are quite different, Hickey et al. [2006] also mix de Bruijn and higher-order syntax, using the de Bruijn representation to carve out a class of representational functions and to define an induction principle for them.

All of these approaches for representing binding as computational functions have the advantage that they can be carried out in existing proof assistants. In contrast, by defining a new type theory, we are able to give a simple account of the abstraction that these constructions are trying achieve. Our representational functions provide a direct tool for adequately encoding binding, and their elimination form gives structural recursion on open terms. Moreover, as we hope to have demonstrated, $\rightarrow$ and $\Rightarrow$ are really quite orthogonal connectives, and encoding one in terms of the other ignores some of their essential properties (such as, for example, the distributivity principles in Section 3.1).

Miller [1990] shared this stance, and proposed an extension to ML with a new type `'a => 'b` representing a term of type `'b` with abstracted parameter `'a`, as well as a restricted form of higher-order pattern-matching. Although it came long before the proof-theoretic innovations that made our work possible, Miller's proposal has several interesting features that can be reexamined in light of our analysis. For example, the domain `'a` must be not only an equality type, but also a user-defined datatype, since the meanings of base types such as `int` or `string` should not be open-ended. This is related to (although less general than) the present restriction that the domain of $R \Rightarrow A^+$ be a rule extending the meaning of a defined atom P (though R need not be P itself). The fact that the codomain `'b` must be an equality type in Miller's proposal is related to (although less general than) the present restriction that the codomain A^+ be positive (though 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's more recent work [Miller and Tiu, 2003], as well as in contextual modal type theory [Nanevski et al., 2007]. Indeed, Miller and Tiu's proof theory bears many similarities with ours, although their overall approach is based on the computation-as-proof-search paradigm (i.e., logic programming), whereas ours is based on a proofs-as-programs interpretation of focusing [Zeilberger, 2007], using polarity to segregate computation from data.

Fiore et al. [1999] and Hofmann [1999] 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.

6 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 a practical implementation of our language following the plan sketched in Section 4.2. Additionally, a generalization to dependent types (in which both $\rightarrow$ and $\Rightarrow$ would become dependent function spaces) would realize the goal of primitively supporting higher-order abstract syntax in a constructive type theory, combining the best of frameworks such as Twelf and Coq. Another tantalizing possibility is that the combination of computational open-endedness with higher-order abstract syntax could shed light on the problem of meta-programming. In Section 4.3, we already saw how it was possible to write programs in the object-language using the full power of the meta-language.

Finally, the present paper serves as a case study in *polarized type theory*. We were able to construct a simple and elegant language, solving a traditionally thorny problem, in part because we based our proof-theoretic analysis on polarity and focusing, which provide a general framework for analyzing the interaction of rules independently of particular connectives. In "Locus Solum", Girard proposed this framework as a new approach to the study of logic. We believe this approach also has much potential for practical programming.

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

References

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.

S. Ambler, R. Crole, and A. Momigliano. A definitional approach to primitive recursion over higher order abstract syntax. In *Workshop on Mechanized reasoning about languages with variable binding*. Association for Computing Machinery, June 2003.

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

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.

Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

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.

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

A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5(1):56–68, 1940.

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

T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Lecture Notes in Computer Science*, number 389. Springer-Verlag, 1989.

N. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.

J. Despeyroux and P. Leleu. Primitive recursion for higher-order abstract syntax with dependant types. In *Workshop on Intuitionistic Modal Logics and Applications*, Tento, Italy, 1999.

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.

M. Dummett. *The Logical Basis of Metaphysics*. The William James Lectures, 1976. Harvard University Press, Cambridge, Massachusetts, 1991.

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

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.

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.

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

R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Symposium on Principles of Programming Languages*, 1995.

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

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.

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

D. J. Howe. On computational open-endedness in Martin-Löf's type theory. In *LICS*, pages 162–172. IEEE Computer Society, 1991.

O. Laurent. *Etude de la polarisation en logique*. Thèse de doctorat, Université Aix-Marseille II, Mar. 2002.

C. Liang and D. Miller. Focusing and polarization in intuitionistic logic. In J. Duparc and T. A. Henzinger, editors, *CSL 2007: Computer Science Logic*, volume 4646 of *LNCS*, pages 451–465. Springer-Verlag, 2007.

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.

P. F. Mendler. *Inductive Definition in Type Theory*. Phd thesis (Technical Report TR 87-870), Dept. of Computer Science, Cornell Univ., Ithaca, NY, Sept. 1987.

D. Miller. An extension to ML to handle bound variables in data structures: Preliminary report. Technical report, Pennsylvania State University, Department of Computer Science and Engineering, Aug. 1990.

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.

R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

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.

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

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

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.

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.

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

A. M. Pitts. Alpha-structural recursion and induction. *Journal of the Association for Computing Machinery*, 53:459–506, 2006.

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 Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

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

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

P. Schroeder-Heister. A natural extension of natural deduction. *Journal of Symbolic Logic*, 49(4):1284–1300, 1984.

P. Schroeder-Heister. Cut-elimination in logics with definitional reflection. In D. Pearce and H. Wansing, editors, *Nonclassical Logics and Information Processing*, volume 619 of *Lecture Notes in Computer Science*, pages 146–171. Springer, 1992.

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

P. Sewell, F. Z. Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. In *ACM SIGPLAN International Conference on Functional Programming*, 2007.

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.

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

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.

N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 2007. To appear in a special issue on “Classical Logic and Computation”.

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