Focusing and Higher-Order Abstract Syntax

Noam Zeilberger
Carnegie Mellon University
noam@cs.cmu.edu

Abstract

Focusing is a proof-search strategy, originating in linear logic, that elegantly eliminates inessential nondeterminism, with one byproduct being a correspondence between focusing proofs and programs with explicit evaluation order. Higher-order abstract syntax (HOAS) is a technique for representing higher-order programming language constructs (e.g., λ's) by higher-order terms at the "meta-level", thereby avoiding some of the bureaucratic headaches of first-order representations (e.g., capture-avoiding substitution).

This paper begins with a fresh, judgmental analysis of focusing for intuitionistic logic (with a full suite of propositional connectives), recasting the "derived rules" of focusing as *iterated inductive definitions.* This leads to a uniform presentation, allowing concise, modular proofs of the identity and cut principles. Then we show how this formulation of focusing induces, through the Curry-Howard isomorphism, a new kind of higher-order encoding of abstract syntax: functions are encoded by maps from *patterns* to expressions. Dually, values are encoded as patterns together with *explicit substitutions.* This gives us pattern-matching "for free", and lets us reason about a rich type system with minimal syntactic overhead. We describe how to translate the language and proof of type safety almost directly into Coq using HOAS, and finally, show how the system's modular design pays off in enabling a very simple extension with recursion and recursive types.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory; F.4.1 [*Theory of Computation*]: Mathematical Logic—*Lambda calculus and related systems*

General Terms Languages

1. Introduction

The end result of this paper will be to show how so-called *focusing* proofs produce—through a careful judgmental analysis and the Curry-Howard isomorphism—an exceptionally compact presentation of a call-by-value language with a full suite of types. In the process, we hope to convince the reader of an aphorism: abstract syntax should be even *more* abstract.

The technique of focusing was originally invented by Andreoli (1992) as a refinement of bottom-up proof search in linear logic, to reduce an otherwise intractable amount of nondeterminism. Soon afterwards, it was promoted by Girard (1993) as a conceptual tool for finding unity in logic, as it turned out that also the classical and intuitionistic connectives could be classified by their focusing behavior, or *polarity*. Recently, focusing and polarity have seen a surge in interest as more and more surprising properties of focusing proofs are discovered, including one important example: it is slowly becoming clear that focusing proofs correspond to programs with explicit evaluation order (Herbelin 1995; Curien and Herbelin 2000; Selinger 2001; Laurent 2002; Wadler 2003; Laurent 2005; Dyckhoff and Lengrand 2006). In this paper we will demonstrate an additional fascinating fact about focusing proofs: they correspond to programs with *pattern-matching.* Moreover, it turns out that focusing can be given a uniform, higher-order formulation as an *iterated inductive definition,* and that this representation can be encoded naturally in Coq (Martin-Löf 1971; Coquand and Paulin-Mohring 1989). Combining these facts, we obtain the above aphorism: a new kind of higher-order abstract syntax that encodes "pattern-matching for free".

POPL '08 January 10–12, 2008, San Francisco, CA.

2. Focusing intuitionistic logic

2.1 Background

Before diving into the compact presentation of focusing and its Curry-Howard interpretation à la HOAS, let us start on more familiar ground with a standard intuitionistic sequent calculus, and describe how to obtain a "small-step" focusing system. Figure 1 gives the sequent calculus for intuitionistic logic in a slight variation of Kleene's **G3i** formulation (Kleene 1952; Troelstra and Schwichtenberg 1996). Formulas (P, Q, R) are built out of conjunction ($\times$) and disjunction ($+$) and their respective units (1 and 0), implication ($\rightarrow$), and logical atoms (X, Y, Z). Every logical connective has a pair of a left rule and right rule(s) (we omit the rules for the units to save space). The identity rule is restricted to atoms and there is no explicit cut rule, though both cut (from $\Gamma \vdash P$ and $\Gamma, P \vdash Q$ conclude $\Gamma \vdash Q$) and the general identity principle ($P \in \Gamma$ implies $\Gamma \vdash P$) are admissible.

Now, one way to conceive of the sequent calculus, as Gentzen (1935) originally suggested, is as a proof search procedure. Each rule can be read bottom-up as a prescription, "To prove the conclusion, try proving the premises". Starting from a goal sequent $\Gamma \vdash P$, one attempts to build a proof by invoking left- and right-rules provisionally to obtain a new set of goals until, hopefully, all goals can be discharged using rules with no premises (i.e., *id*, $1R$ or $0L$). Since there are only finitely many rules and each satisfies the *subformula property* (Troelstra and Schwichtenberg 1996), it is not hard to see that (so long as one checks saturation conditions to avoid repeatedly applying left rules) the sequent calculus gives a naive decision procedure for propositional intuitionistic logic.

The reason this decision procedure is naive, though, is because the order of application of rules is left entirely unspecified. For example, the following are two equally legitimate derivations of $X \times Y \vdash X \times Y$, that differ only in the order of $\times L$ and $\times R$:

Context $\Gamma ::= \cdot \mid \Gamma, P$

$$\frac{X \in \Gamma}{\Gamma \vdash X}\ id \qquad \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \to Q}\ {\to}R$$

$$\frac{\Gamma \vdash P \quad \Gamma \vdash Q}{\Gamma \vdash P \times Q}\ {\times}R \qquad \frac{\Gamma \vdash P}{\Gamma \vdash P + Q}\ {+}R \qquad \frac{\Gamma \vdash Q}{\Gamma \vdash P + Q}\ {+}R$$

$$\frac{P \times Q \in \Gamma \quad \Gamma, P, Q \vdash R}{\Gamma \vdash R}\ {\times}L$$

$$\frac{P + Q \in \Gamma \quad \Gamma, P \vdash R \quad \Gamma, Q \vdash R}{\Gamma \vdash R}\ {+}L$$

$$\frac{P \to Q \in \Gamma \quad \Gamma \vdash P \quad \Gamma, Q \vdash R}{\Gamma \vdash R}\ {\to}L$$

Figure 1. Intuitionistic sequent calculus

$$\frac{\dfrac{\overline{X \times Y, X, Y \vdash X}\ id \quad \overline{X \times Y, X, Y \vdash Y}\ id}{X \times Y, X, Y \vdash X \times Y}\ {\times}R}{X \times Y \vdash X \times Y}\ {\times}L$$

$$\frac{\dfrac{\dfrac{\overline{X \times Y, X, Y \vdash X}\ id}{X \times Y \vdash X}\ {\times}L \quad \dfrac{\overline{X \times Y, X, Y \vdash Y}\ id}{X \times Y \vdash Y}\ {\times}L}{X \times Y \vdash X \times Y}\ {\times}R$$

However, it is not the case that order of application is *arbitrary.* For example, to prove $X + Y \vdash X + Y$, one must apply $+L$ first (from the bottom):

$$\frac{\dfrac{\overline{X + Y, X \vdash X}\ id}{X + Y, X \vdash X + Y}\ {+}R \quad \dfrac{\overline{X + Y, Y \vdash Y}\ id}{X + Y, Y \vdash X + Y}\ {+}R}{X + Y \vdash X + Y}\ {+}L$$

Applying either right-rule first will yield a failed proof attempt.

In these terms, focusing can be seen as exploiting properties about the connectives to implement a smarter bottom-up proof search. Figure 2 presents a focusing system for intuitionistic logic that implements the following strategy:

1. Decompose conjunctions and disjunctions greedily on the left, until the context contains only atoms and implications.
2. Given a stable sequent (i.e., one with no undecomposed hypotheses), "focus" on some proposition ($\Gamma \vdash [P]$), either the right side of the sequent or the antecedent of a hypothesis $P \to Q$.
3. A proposition in focus remains in focus (forcing us to keep applying right-rules) until either there are no more premises, or else we reach an implication, which "blurs" the sequent (and we go back to step 1).

Note that this is not the only possible focusing strategy for propositional intuitionistic logic. Most of the intuitionistic connectives have ambiguous *polarity,* in Girard's sense (Girard 1993). This is in contrast with the connectives of linear logic, which have fixed polarity. So whereas there is essentially only one way to focus linear logic, there are different possible strategies for intuitionistic logic, corresponding to different *polarizations.* Our strategy treats conjunction and disjunction as both *positive,* implication as *negative,* which turns out to correspond, via Curry-Howard, to the strict, call-by-value interpretation (Curien and Herbelin 2000; Selinger 2001; Laurent 2005). To emphasize this fact, we adopt linear logic notation for positive conjunction and disjunction ($\otimes$, $\oplus$, 1, 0), and

Stable context $\Gamma ::= \cdot \mid \Gamma, X \mid \Gamma, P \xrightarrow{v} Q$
Active context $\Omega ::= \cdot \mid P, \Omega$

$\boxed{\Gamma \vdash [P]}$

$$\frac{X \in \Gamma}{\Gamma \vdash [X]} \qquad \frac{\Gamma; P \vdash Q}{\Gamma \vdash [P \xrightarrow{v} Q]}$$

$$\frac{\Gamma \vdash [P] \quad \Gamma \vdash [Q]}{\Gamma \vdash [P \otimes Q]} \qquad \frac{\Gamma \vdash [P]}{\Gamma \vdash [P \oplus Q]} \qquad \frac{\Gamma \vdash [Q]}{\Gamma \vdash [P \oplus Q]}$$

$\boxed{\Gamma; \Omega \vdash R}$

$$\frac{\Gamma, X; \Omega \vdash R}{\Gamma; X, \Omega \vdash R} \qquad \frac{\Gamma, P \xrightarrow{v} Q; \Omega \vdash R}{\Gamma; P \xrightarrow{v} Q, \Omega \vdash R}$$

$$\frac{\Gamma; P, Q, \Omega \vdash R}{\Gamma; P \otimes Q, \Omega \vdash R} \qquad \frac{\Gamma; P, \Omega \vdash R \quad \Gamma; Q, \Omega \vdash R}{\Gamma; P \oplus Q, \Omega \vdash R}$$

$$\frac{\Gamma \vdash R}{\Gamma; \cdot \vdash R}$$

$\boxed{\Gamma \vdash R}$

$$\frac{\Gamma \vdash [P]}{\Gamma \vdash P} \qquad \frac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash [P] \quad \Gamma; Q \vdash R}{\Gamma \vdash R}$$

Figure 2. Focused intuitionistic sequent calculus

write $\xrightarrow{v}$ for implication.[1] Very similar focusing systems based on the same polarizations are presented in (Girard 2001, §9.2.3) and (Dyckhoff and Lengrand 2006). Some examples of focusing systems derived from alternative (lazy, call-by-name) polarizations of intuitionistic logic are in (Herbelin 1995; Howe 1998; Miller and Liang 2007).

Of course, from the point of view of proof-search, it is crucial that any focusing strategy be *complete,* i.e., if a sequent is provable in the ordinary sequent calculus, then the focusing strategy will succeed in finding *some* derivation. We will not give a completeness proof for this system here (the reader could refer to (Dyckhoff and Lengrand 2006)), and instead move on to describe an alternative presentation of focusing.

2.2 A higher-order formulation

Let us begin with some observations about derived rules in the focused system. These observations are not new (Andreoli 2001; Girard 2001)—but the system we obtain from these observations will be.

Consider proving the proposition $X \otimes (Y \oplus (P \xrightarrow{v} Q))$ in focus. The derivation must begin in one of the following two ways, before losing focus:

$$\frac{\Gamma \vdash [X] \quad \dfrac{\Gamma \vdash [Y]}{\Gamma \vdash [Y \oplus (Q \xrightarrow{v} P)]}}{\Gamma \vdash [X \otimes (Y \oplus (Q \xrightarrow{v} P))]} \qquad \frac{\Gamma \vdash [X] \quad \dfrac{\Gamma \vdash [P \xrightarrow{v} Q]}{\Gamma \vdash [Y \oplus (P \xrightarrow{v} Q)]}}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q))]}$$

Once in focus, atomic propositions can only be proven by assumption, while implications initiate a decomposition phase. The set of derived rules

$$\frac{X \in \Gamma \quad Y \in \Gamma}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q))]} \qquad \frac{X \in \Gamma \quad \Gamma; P \vdash Q}{\Gamma \vdash [X \otimes (Y \oplus (P \xrightarrow{v} Q)))]}$$

[1] Describing the polarity of call-by-value implication is actually a bit more subtle. Technically, one can identify an underlying negative implication $P \multimap N$ which takes positive antecedent and negative consequent, and then analyze $P \xrightarrow{v} Q$ with implicit polarity "shifts" (Girard 2001, §3.3.2), i.e., either as $P \multimap \uparrow Q$ (as a negative hypothesis) or $\downarrow(P \multimap \uparrow Q)$ (as a positive conclusion).

is therefore *complete,* in the sense that it covers all possible derivations of the formula in right-focus.

Similarly, consider decomposing $X \otimes (Y \oplus (P \xrightarrow{v} Q))$ on the left of the sequent:

$$\dfrac{\dfrac{\dfrac{\Gamma, X, Y \vdash R}{\Gamma, X, Y; \cdot \vdash R}}{\Gamma, X; Y \vdash R} \quad \dfrac{\dfrac{\Gamma, X, P \xrightarrow{v} Q \vdash R}{\Gamma, X, P \xrightarrow{v} Q; \cdot \vdash R}}{\Gamma, X; P \xrightarrow{v} Q \vdash R}}{\dfrac{\Gamma, X; Y \oplus (P \xrightarrow{v} Q) \vdash R}{\dfrac{\Gamma; X, Y \oplus (P \xrightarrow{v} Q) \vdash R}{\Gamma; X \otimes (Y \oplus (P \xrightarrow{v} Q)) \vdash R}}}$$

Again, the following derived rule is complete:

$$\dfrac{\Gamma, X, Y \vdash R \quad \Gamma, X, P \xrightarrow{v} Q \vdash R}{\Gamma; X \otimes (Y \oplus (P \xrightarrow{v} Q)) \vdash R}$$

In general for a proposition P, we can give a complete set (possibly empty) of derived rules for establishing $\Gamma \vdash [P]$, each containing a set (possibly empty) of premises of the form $X \in \Gamma$ or $\Gamma; Q \vdash R$. Likewise, we can give a single, complete derived rule for establishing $\Gamma; P \vdash R$, with a set (possibly empty) of premises of the form $\Gamma, \Gamma' \vdash R$.

Both kinds of derived rules for a formula P can be generated from a single description, which we can gloss as the possible "recipes" for a focused proof. To derive $\Gamma \vdash [P]$, we must provide (using Γ) all of the "ingredients" for *some* recipe. To derive $\Gamma; P \vdash R$, we must show how to derive R given (Γ and) the ingredients for *any* of the recipes. As we are trying to suggest by using culinary language (Wadler 1993), the method for constructing both kinds of derived rules can be expressed in terms of linear entailment.

More precisely, a "list of ingredients" Δ is a linear context of atoms and implications, and we write $\Delta \Rightarrow P$ when Δ exactly describes the focused premises in a possible focused proof of P. The rules for $\Delta \Rightarrow P$ (top of Figure 3) are just the usual right-rules for the positive connectives of linear logic together with axioms $X \Rightarrow X$ and $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$. By way of example, we have $X, Y \Rightarrow X \otimes (Y \oplus (P \xrightarrow{v} Q))$ and $X, P \xrightarrow{v} Q \Rightarrow X \otimes (Y \oplus (P \xrightarrow{v} Q))$. Note that the judgment $\Delta \Rightarrow P$ obeys a subformula property.

Proposition (Subformula property)**.** *If $\Delta \Rightarrow P$, then Δ contains only subformulas of P.*

The generic instructions for proving a proposition in focus, which we described informally above, can now be written formally:

$$\dfrac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

The judgment $\Gamma \vdash \Delta$ is interpreted conjunctively:[2] from the hypotheses in Γ we must prove everything in Δ. Thus the rule asks for some choice of recipe (i.e., $\Delta \Rightarrow P$), and a proof that we have all the ingredients (i.e., $\Gamma \vdash \Delta$). Note that although this rule leaves Δ unspecified, it still obeys the usual subformula property, by the subformula property for $\Delta \Rightarrow P$.

Likewise, we can write the generic rule for decomposing a proposition on the left:

$$\dfrac{\forall(\Delta \Rightarrow P) : \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

Here the rule quantifies over *all* Δ such that $\Delta \Rightarrow P$, showing that from any such Δ (together with Γ), Q is derivable. This sort of quantification over derivations might seem like a risky form of definition, but it is simply an *iterated* inductive definition (Martin-Löf 1971)—since we already established what $\Delta \Rightarrow P$ means, there is no circularity in treating it as an assumption here. Indeed, for any particular P built out of the connectives we have considered, there will only ever be finitely many derivations $\Delta \Rightarrow P$, so this rule will just have a finite list of premises (as in the example above). However, we hope to make the case that this higher-order formulation should be taken at face value—interpreted constructively, it demands a *mapping* from derivations of $\Delta \Rightarrow P$ to unfocused sequents $\Gamma, \Delta \vdash Q$. This idea will play a central role in our Curry-Howard interpretation.

[2] And so is *not* like a "multiple conclusion sequent" in Gentzen's LK.

Linear context $\Delta ::= \cdot \mid X, \Delta \mid P \xrightarrow{v} Q, \Delta$

$\boxed{\Delta \Rightarrow P}$

$$\dfrac{}{X \Rightarrow X} \qquad \dfrac{}{P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q}$$

$$\dfrac{}{\cdot \Rightarrow 1} \qquad \dfrac{\Delta_1 \Rightarrow P \quad \Delta_2 \Rightarrow Q}{\Delta_1, \Delta_2 \Rightarrow P \otimes Q}$$

$$\text{(no rule for 0)} \qquad \dfrac{\Delta \Rightarrow P}{\Delta \Rightarrow P \oplus Q} \qquad \dfrac{\Delta \Rightarrow Q}{\Delta \Rightarrow P \oplus Q}$$

Stable context $\Gamma ::= \cdot \mid \Gamma, \Delta$

$\boxed{\Gamma \vdash [P]}$

$$\dfrac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

$\boxed{\Gamma; P \vdash Q}$

$$\dfrac{\forall(\Delta \Rightarrow P) : \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

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

$$\dfrac{}{\Gamma \vdash \cdot} \qquad \dfrac{X \in \Gamma \quad \Gamma \vdash \Delta}{\Gamma \vdash X, \Delta} \qquad \dfrac{\Gamma; P \vdash Q \quad \Gamma \vdash \Delta}{\Gamma \vdash P \xrightarrow{v} Q, \Delta}$$

$\boxed{\Gamma \vdash R}$

$$\dfrac{\Gamma \vdash [P]}{\Gamma \vdash P} \qquad \dfrac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash [P] \quad \Gamma; Q \vdash R}{\Gamma \vdash R}$$

Figure 3. Large-step focusing

The entire "large-step" focusing system is given in Figure 3, with all of the rules for all of the connectives (including the units 1 and 0). Observe that the *only* rules that explicitly mention the positive connectives are those for the $\Delta \Rightarrow P$ judgment, and we can take the latter as literally *defining* the positive connectives. While the system is relatively sparse in rules, it is "rich in judgments". The idea of the *judgmental method* (Martin-Löf 1996; Pfenning and Davies 2001) in general is that by distinguishing between different kinds of reasoning as different judgments (and not merely between different logical connectives or type constructors), one can clarify the structure of proofs. This becomes very vivid under a Curry-Howard interpretation, as the proofs of different judgments are internalized by different syntactic categories of a programming language. We will find that the five judgments of large-step focusing all correspond to very natural programming constructs. First, though, let us see how the identity and cut principles work in this new logical setting. Because of the additional judgmental machinery, identity is refined into three different principles.

Principle (Identity)**.** $\Gamma; P \vdash P$

Principle (Context identity)**.** *If $\Gamma \supseteq \Delta$ then $\Gamma \vdash \Delta$*

Principle (Arrow identity)**.** *If $P \xrightarrow{v} Q \in \Gamma$ then $\Gamma; P \vdash Q$*

Proof. These three principles are proven simultaneously—we give the proof first, and then explain its inductive structure.

- (Identity) The following derivation reduces identity to context identity:

$$\dfrac{\forall(\Delta \Rightarrow P): \quad \dfrac{\dfrac{\Delta \Rightarrow P \quad \Gamma, \Delta \vdash \Delta}{\Gamma, \Delta \vdash [P]}}{\Gamma, \Delta \vdash P}}{\Gamma; P \vdash P}$$

  Note the first premise can be discharged since the derivation quantifies over Δ such that $\Delta \Rightarrow P$.
- (Context Identity) We apply a side-induction on the length of Δ. The interesting case is $\Delta = P \xrightarrow{v} Q, \Delta'$. By arrow identity we have $\Gamma; P \vdash Q$, and by the side-induction we have $\Gamma \vdash \Delta'$, letting us build the derivation:

$$\dfrac{\Gamma; P \vdash Q \quad \Gamma \vdash \Delta'}{\Gamma \vdash P \xrightarrow{v} Q, \Delta'}$$

- (Arrow Identity) Consider the following derivation:

$$\dfrac{\forall(\Delta \Rightarrow P): \quad \dfrac{P \xrightarrow{v} Q \in \Gamma \quad \Gamma, \Delta \vdash [P] \quad \Gamma, \Delta; Q \vdash Q}{\Gamma, \Delta \vdash Q}}{\Gamma; P \vdash Q}$$

  The first premise $P \xrightarrow{v} Q \in \Gamma$ is by assumption. The second premise reduces (as in the proof of identity above) to context identity. The third premise is by identity.

The above argument can be seen to be well-founded so long as the relationship of being a proper subformula is well-founded. We reason as follows: The proof of identity appealed to context identity, which in turn appealed to arrow identity, and which finally appealed back to both context identity and identity. The first cycle (id on $P \rightsquigarrow$ context id on $\Delta \Rightarrow P \rightsquigarrow$ arrow id on $P_1 \xrightarrow{v} P_2 \in \Delta \rightsquigarrow$ id on P_2), takes P to a proper subformula P_2. The second cycle (context id on $\Delta \rightsquigarrow$ arrow id on $P_1 \xrightarrow{v} P_2 \in \Delta \rightsquigarrow$ context id on $\Delta' \Rightarrow P_1$), takes Δ to a proper *subcontext* Δ' (i.e., Δ' contains only proper subformulas of formulas in Δ). Both cycles cannot continue indefinitely if the proper subformula relationship is well-founded—as indeed it is for the propositional connectives. □

We can also distinguish between three different kinds of principles that would ordinarily be called "cuts". The first is where we have a derivation of $\Gamma, \Delta \vdash J$ (in which J stands for an arbitrary concluding judgment, i.e., $\Gamma, \Delta \vdash [P]$ or $\Gamma, \Delta; P \vdash Q$ or $\Gamma, \Delta \vdash R$ or $\Gamma, \Delta \vdash \Delta'$), and we want to substitute another derivation $\Gamma \vdash \Delta$ for the hypotheses Δ. The second is where we have a coincidence between a right-focused derivation $\Gamma \vdash [P]$, and a derivation $\Gamma; P \vdash Q$, which we can transform into an unfocused derivation $\Gamma \vdash Q$. In the third, we combine an unfocused derivation $\Gamma \vdash P$ together with $\Gamma; P \vdash Q$ to obtain $\Gamma \vdash Q$. We call the first cut principle *substitution,* the second *reduction,* and the third *composition.* In the usual proof-theoretic terminology, these correspond to *right-commutative*, *principal,* and *left-commutative cuts,* respectively.

Principle (Substitution)**.** *If* $\Gamma, \Gamma' \vdash \Delta$ *and* $\Gamma, \Delta, \Gamma' \vdash J$ *then* $\Gamma, \Gamma' \vdash J$

Principle (Reduction)**.** *If* $\Gamma \vdash [P]$ *and* $\Gamma; P \vdash Q$ *then* $\Gamma \vdash Q$

Principle (Composition)**.** *If* $\Gamma \vdash P$ *and* $\Gamma; P \vdash Q$ *then* $\Gamma \vdash Q$

To prove these we need a weakening lemma, which is immediate.

Proposition (Weakening)**.** *If* $\Gamma \vdash J$*, then* $\Gamma, \Delta \vdash J$.

Proof of substitution, reduction and composition. Again, the proof is simultaneous.

- (Substitution) We examine the derivation of $\Gamma, \Delta, \Gamma' \vdash J$. Almost all cases (there are seven total) are immediate, simply applying substitution (and possibly weakening) to the premises and reconstructing the derivation. The one interesting case is the following:

$$\dfrac{P \xrightarrow{v} Q \in \Delta \quad \Gamma, \Delta, \Gamma' \vdash [P] \quad \Gamma, \Delta, \Gamma'; Q \vdash R}{\Gamma, \Delta, \Gamma' \vdash R}$$

  By substitution on the premises we have $\Gamma, \Gamma' \vdash [P]$ and $\Gamma, \Gamma'; Q \vdash R$. Moreover $\Gamma, \Gamma' \vdash \Delta$ and $P \xrightarrow{v} Q \in \Delta$ imply (by inversion) that $\Gamma, \Gamma'; P \vdash Q$. We cut $\Gamma, \Gamma' \vdash [P]$ and $\Gamma, \Gamma'; P \vdash Q$ using reduction to obtain $\Gamma, \Gamma' \vdash Q$, and the latter with $\Gamma, \Gamma'; Q \vdash R$ using composition to obtain $\Gamma, \Gamma' \vdash R$.
- (Reduction) By inversion on $\Gamma \vdash [P]$, there exists some $\Delta \Rightarrow P$ such that $\Gamma \vdash \Delta$, and by inversion on $\Gamma; P \vdash Q$ we have $\Gamma, \Delta \vdash Q$. Hence $\Gamma \vdash Q$ by substitution.
- (Composition) We examine the derivation of $\Gamma \vdash P$. If it was derived from $\Gamma \vdash [P]$, we immediately apply reduction. Otherwise the derivation must look like so:

$$\dfrac{P_1 \xrightarrow{v} P_2 \in \Gamma \quad \Gamma \vdash [P_1] \qquad \dfrac{\forall(\Delta \Rightarrow P_2): \quad \Gamma, \Delta \vdash P}{\Gamma; P_2 \vdash P}}{\Gamma \vdash P}$$

  For any $\Delta \Rightarrow P_2$, we can weaken the derivation $\Gamma; P \vdash Q$ to $\Gamma, \Delta; P \vdash Q$, and then apply composition to obtain $\Gamma, \Delta \vdash Q$. Thus $\Gamma; P_2 \vdash Q$, and we can reconstruct the derivation concluding $\Gamma \vdash Q$.

The above defines a cut-elimination procedure, which we can easily see is terminating by a nested induction. First on the cut formula/context, then on the second derivation for substitution, and on the first derivation for composition. Again, this uses the fact that the proper subformula relationship is well-founded. □

We gave the proofs of identity and cut in such explicit detail in part to emphasize that there actually *isn't* very much detail. For example, we did not have to give the case of one "typical" positive connective and sweep the others under the rug, because both proofs do not even *mention* particular positive connectives—instead they reason modularly about derivations of $\Delta \Rightarrow P$. And modularity is a powerful tool: it gives us license to introduce new types almost arbitrarily, so long as we define them purely through the $\Delta \Rightarrow P$ judgment.

3. Focusing the λ-calculus

In the previous section, we saw how combining the technique of focusing with a judgmental and higher-order analysis of derived rules led to a sequent calculus "rich in judgments". Now, we will show how these different judgments correspond precisely, through the Curry-Howard isomorphism, to natural programming language constructs. We start with a type system containing all the propositional connectives described above, though for simplicity leaving out atomic types—so the language will have strict products and sums, and call-by-value function spaces. After giving it an operational semantics corresponding to cut-elimination and proving type safety, we will show how our informal use of higher-order abstract syntax can be formalized in Coq. Finally, we will try to give a demonstration of the aforementioned modularity principle, by showing the ease with which recursion and recursive types can be added to the language.

Focusing	Typing	Syntactic category
$\Delta \Rightarrow P$	$\Delta \Rightarrow p : P$	patterns
$\Gamma \vdash [P]$	$\Gamma \vdash V : [P]$	values
$\Gamma; P \vdash Q$	$\Gamma \vdash F : P > Q$	(CBV) functions
$\Gamma \vdash R$	$\Gamma \vdash E : R$	expressions
$\Gamma \vdash \Delta$	$\Gamma \vdash \sigma : \Delta$	substitutions

Figure 4. The Curry-Howard isomorphism

3.1 Type system

Let us begin by examining the $\Delta \Rightarrow P$ judgment, which lies at the heart of our formulation of focusing. Previously we described $\Delta \Rightarrow P$ as holding when the context Δ is an exact list of focused premises needed for a focused proof of P. For a particular P, there need not be a unique Δ such that $\Delta \Rightarrow P$, and indeed there might not be any such Δ (e.g., when $P = 0$). What then do derivations of $\Delta \Rightarrow P$ look like? Abstractly, they describe the different *shapes* a focused proof of P can have, up to the point where either the derivation ends or focus is lost. Thus for example we only have the axiomatic derivation $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$, because the first step in a focused proof of $P \xrightarrow{v} Q$ is to immediately lose focus. On the other hand, there are two rules for disjunction:

$$\frac{\Delta \Rightarrow P}{\Delta \Rightarrow P \oplus Q} \quad \frac{\Delta \Rightarrow Q}{\Delta \Rightarrow P \oplus Q}$$

because a focused proof of $P \oplus Q$ can continue by focusing on either P or Q.

Now, let us label the hypotheses in Δ with variables—since we are ignoring atomic hypotheses, there are only function variable hypotheses $f : P \xrightarrow{v} Q$. Then we can annotate $\Delta \Rightarrow P$ as a *pattern-typing* judgment:

$$\frac{}{f : P \xrightarrow{v} Q \Rightarrow f : P \xrightarrow{v} Q}$$

$$\frac{}{\cdot \Rightarrow () : 1} \qquad \frac{\Delta_1 \Rightarrow p_1 : P \quad \Delta_2 \Rightarrow p_2 : Q}{\Delta_1, \Delta_2 \Rightarrow (p_1, p_2) : P \otimes Q}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Rightarrow p : P}{\Delta \Rightarrow \mathsf{inl}\ p : P \oplus Q} \quad \frac{\Delta \Rightarrow p : Q}{\Delta \Rightarrow \mathsf{inr}\ p : P \oplus Q}$$

A programmer might now get an intuition for why the context Δ must be linear: it corresponds to the usual restriction that *patterns cannot bind a variable more than once.* Likewise why $P \xrightarrow{v} Q \Rightarrow P \xrightarrow{v} Q$ is an axiom: it corresponds to a *primitive pattern.*

If $\Delta \Rightarrow P$ represents pattern-typing, what can we conclude about the other judgments of the focusing system? As we will describe, these correspond to typing judgments for *values, functions, expressions,* and *substitutions* (see Figure 4). Since these judgments are defined by mutual recursion, we will have to work our way through the system to convince ourselves that these names for the different syntactic categories were not chosen arbitrarily. We begin with $\Gamma \vdash [P]$, which will be annotated with a value V. Recall that the judgment is defined by a single rule:

$$\frac{\Delta \Rightarrow P \quad \Gamma \vdash \Delta}{\Gamma \vdash [P]}$$

The first premise is now annotated $\Delta \Rightarrow p : P$, giving us a pattern p binding some function variables with types given by Δ. The second premise is annotated with a simultaneous substitution $\sigma = (F_1/f_1, \ldots, F_n/f_n)$, where $f_1, \ldots, f_n$ are the variables in Δ and $F_1, \ldots, F_n$ are functions. Thus the annotated rule becomes:

$$\frac{\Delta \Rightarrow p : P \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash [\sigma]p : [P]}$$

What exactly is this curious value $[\sigma]p$, which combines a pattern together with an explicit substitution? We can think of this notation as simply internalizing a trivial factorization lemma about values in the ordinary sense. For example the ML value

```
(fn x => x*x, fn x => x-3)
```

can be factored as the pattern `(f,g)` composed with a substitution $[$`(fn x => x*x)`$/$`f`, `(fn x => x-3)`$/$`g`$]$. As we shall see, the utility of this factorization is that values are given a uniform representation.

What about functions? Again, let us look at the unannotated rule for $\Gamma; P \vdash Q$:

$$\frac{\forall(\Delta \Rightarrow P) : \quad \Gamma, \Delta \vdash Q}{\Gamma; P \vdash Q}$$

Recall this is a higher-order rule, which can be interpreted constructively as demanding a map from derivations of $\Delta \Rightarrow P$ to derivations of unfocused sequents $\Gamma, \Delta \vdash Q$. The former, we know, correspond to patterns with types for their free variables. The latter correspond to "expressions" (the precise sense of which will be explained below). Therefore, a function is *a map from patterns to expressions.* In other words, functions are defined using higher-order abstract syntax (Pfenning and Elliott 1988).

Formally, we will assume the existence of partial maps ϕ, defined inductively over patterns. Thus for any pattern p, $\phi(p)$ is either undefined or denotes a unique expression, possibly mentioning variables bound by p, and moreover this mapping respects renaming of pattern variables. Concretely, partial maps may be specified by a finite list of branches:

$$\phi ::= (p_1 \mapsto E_1 \mid \ldots \mid p_n \mapsto E_n)$$

with the proviso that the p_i do not overlap. In Section 3.3 we will describe how to encode the HOAS representation explicitly in Coq, using the function space $\mathsf{pat} \to \mathsf{exp}$.

Now to build a function, we simply wrap a ϕ with a λ. The annotated rule for function-typing becomes:

$$\frac{\forall(\Delta \Rightarrow p : P) : \quad \Gamma, \Delta \vdash \phi(p) : Q}{\Gamma \vdash (\lambda\phi) : P > Q}$$

We should emphasize that we are still only defining the abstract *syntax* of functions, not their evaluation semantics—although the two aspects are indeed closely related. For instance, the syntax forces a call-by-value interpretation, since functions are defined by pattern-matching over fully-expanded patterns.[3] Moreover, a well-typed function $(\lambda\phi) : P > Q$ is necessarily *exhaustive* (since the typing rule forces $\phi(p)$ to be defined for all p and Δ such that $\Delta \Rightarrow p : P$) and *non-redundant* (since ϕ is defined as a map), in the usual sense of pattern-matching.

Finally, the two rules for deriving unfocused sequents are now annotated as typing expressions:

$$\frac{\Gamma \vdash V : [P]}{\Gamma \vdash V : P} \qquad \frac{g : P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash V : [P] \quad \Gamma \vdash F : Q; R}{\Gamma \vdash F(g(V)) : R}$$

The first rule creates an expression directly from a value, the second by feeding a value to a named function variable, and composing the result with another function. From these two rules, we can intuit that "expressions" really do correspond closely to expressions in the ML sense—that is to *computations* (Moggi 1991). However, our expressions have a more rigid syntax, with an explicit sequencing of evaluation that resembles A-normal form (Flanagan et al. 1993).

[3] Of course, Haskell has pattern-matching too, so the emphasis is on "fully-expanded". In Haskell, there is a semantic difference between a function defined using wildcard/variable patterns, and the one obtained by replacing the wildcards/variables with expanded patterns. For example `\x -> ()` and `\() -> ()` both can be given type `() -> ()`, but the latter is strict. This difference does not exist in ML since all functions are strict.

Indeed, as with A-normal form, we seem to encounter the problem that substitution requires a "re-normalization" step: for how do we express the result of substituting G/g into $F(g(V))$?

Another way of looking at this is that the expression $F(g(V))$ corresponds to the "one interesting case" in the proof of the substitution principle from Section 2.2, wherein we appealed back to the reduction and composition principles. Consequently, to make the language closed under ordinary substitution, we *internalize* these principles as additional rules for forming expressions:

$$\frac{\Gamma \vdash V : [P] \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(V) : Q} \qquad \frac{\Gamma \vdash E : P \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(E) : Q}$$

We can likewise internalize identity principles to let us take shortcuts when building terms:

$$\frac{}{\Gamma \vdash \mathsf{id} : P > P} \qquad \frac{f : P \xrightarrow{v} Q \in \Gamma}{\Gamma \vdash f : P > Q}$$

id is the polymorphic identity function, while arrow identity allows us to treat a function variable directly as a function.

The complete type system is summarized in Figure 5, defining this Curry-Howard interpretation for large-step focusing, which we call focused λ-calculus. The figure visually quarantines identity and cut principles, to highlight their special status. The reader may wonder why we have not also internalized the substitution and context identity principles. Such steps are possible—for example, internalizing substitution would give us a calculus in which explicit substitutions are evaluated incrementally (Abadi et al. 1991)—but we forgo them here, choosing instead to define these as meta-theoretic operations. Substitution is defined in Section 3.2; we state context identity here:

Principle (Context identity)**.** *Suppose* $\Gamma \supseteq \Delta$*, and that* $f_1, \ldots, f_n$ *are the variables in* Δ*. Then* $\Gamma \vdash (f_1/f_1, \ldots, f_n/f_n) : \Delta$*.*

Proof. Trivial (now that we can directly appeal to arrow identity). □

Let us consider some examples—but to make these more palatable, we first develop some syntactic sugar. Without danger of ambiguity, we can write values in "unfactorized" form:

$$V ::= F \mid () \mid (V_1, V_2) \mid \mathsf{inl}(V) \mid \mathsf{inr}(V)$$

It is always possible to recover a unique factorization, i.e., σ and p such that $V = [\sigma]p$. As a special case, every pattern p can also be seen as the value $[(f_1/f_1, \ldots, f_n/f_n)]p$, where $f_1, \ldots, f_n$ are the variables bound by p. Because the syntax is higher-order, we can use *meta*-variables to build maps by quantifying over (all or some subset of) patterns. So for example $p \mapsto ()$ is a constant map which sends any pattern to (), while the map $f \mapsto ()$ is only defined on function variable patterns. When a function is defined by a single pattern-branch, we use the more conventional notation $\lambda p.E$ instead of $\lambda(p \mapsto E)$. Finally, we let $2 = 1 \oplus 1$ be the type of booleans, write $\mathsf{t} = \mathsf{inl}()$, $\mathsf{f} = \mathsf{inr}()$ for boolean patterns, and use b as a meta-variable quantifying over these two patterns.

EXAMPLE 1. We define boolean functions and and not:

$$\mathsf{and} = \lambda((\mathsf{t},\mathsf{t}) \mapsto \mathsf{t} \mid (\mathsf{t},\mathsf{f}) \mapsto \mathsf{f} \mid (\mathsf{f},\mathsf{t}) \mapsto \mathsf{f} \mid (\mathsf{f},\mathsf{f}) \mapsto \mathsf{f})$$

$$\mathsf{not} = \lambda(\mathsf{t} \mapsto \mathsf{f} \mid \mathsf{f} \mapsto \mathsf{t})$$

It is easy to check that $\mathsf{and} : 2 \otimes 2 > 2$ and $\mathsf{not} : 2 > 2$. In this simple case there is a bijective correspondence between patterns and values, and so the syntax basically mimics the standard mathematical definitions. ■

Linear context $\Delta ::= \cdot \mid f : P \xrightarrow{v} Q, \Delta$
Pattern $p ::= f \mid () \mid (p_1, p_2) \mid \mathsf{inl}\ p \mid \mathsf{inr}\ p$

$\boxed{\Delta \Rightarrow p : P}$

$$\frac{}{f : P \xrightarrow{v} Q \Rightarrow f : P \xrightarrow{v} Q} \qquad \frac{}{\cdot \Rightarrow () : 1} \qquad \frac{\Delta_1 \Rightarrow p_1 : P \quad \Delta_2 \Rightarrow p_2 : Q}{\Delta_1, \Delta_2 \Rightarrow (p_1, p_2) : P \otimes Q}$$

(no rule for 0) $\qquad \frac{\Delta \Rightarrow p : P}{\Delta \Rightarrow \mathsf{inl}\ p : P \oplus Q} \qquad \frac{\Delta \Rightarrow p : Q}{\Delta \Rightarrow \mathsf{inr}\ p : P \oplus Q}$

Stable context $\Gamma ::= \cdot \mid \Gamma, \Delta$
Value $V ::= [\sigma]p$
Function $F ::= \lambda\phi \mid \mathsf{id} \mid f$
where $\phi ::= (p_1 \mapsto E_1 \mid \ldots \mid p_n \mapsto E_n)$
Substitution $\sigma ::= \cdot \mid (F/f, \sigma)$
Expression $E ::= V \mid F(g(V)) \mid F(V) \mid F(E)$

$\boxed{\Gamma \vdash V : [P]}$

$$\frac{\Delta \Rightarrow p : P \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash [\sigma]p : [P]}$$

$\boxed{\Gamma \vdash F : P > Q}$

$$\frac{\forall(\Delta \Rightarrow p : P) :\ \Gamma, \Delta \vdash \phi(p) : Q}{\Gamma \vdash (\lambda\phi) : P > Q} \qquad \frac{}{\Gamma \vdash \mathsf{id} : P > P} \qquad \frac{f : P \xrightarrow{v} Q \in \Gamma}{\Gamma \vdash f : P > Q}$$

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

$$\frac{}{\Gamma \vdash \cdot : \cdot} \qquad \frac{\Gamma \vdash F : P > Q \quad \Gamma \vdash \sigma : \Delta}{\Gamma \vdash (F/f, \sigma) : (f : P \xrightarrow{v} Q, \Delta)}$$

$\boxed{\Gamma \vdash E : R}$

$$\frac{\Gamma \vdash V : [P]}{\Gamma \vdash V : P} \qquad \frac{g : P \xrightarrow{v} Q \in \Gamma \quad \Gamma \vdash V : [P] \quad \Gamma \vdash F : Q; R}{\Gamma \vdash F(g(V)) : R}$$

$$\frac{\Gamma \vdash V : [P] \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(V) : Q} \qquad \frac{\Gamma \vdash E : P \quad \Gamma \vdash F : P > Q}{\Gamma \vdash F(E) : Q}$$

identity principles — cut principles

Figure 5. Focused λ-calculus (type system)

EXAMPLE 2. We define $\mathsf{table1} : 2 \xrightarrow{v} 2 > 2 \otimes 2$, a higher-order function taking a unary boolean operator as input, and returning its truth table as output:

$$\mathsf{table1} = \lambda f.(\lambda b_1.(\lambda b_2.(b_1, b_2))(f\ \mathsf{f}))(f\ \mathsf{t})$$

Here f is a function variable, while b_1 and b_2 are meta-variables quantifying over boolean patterns. Observe that the syntax forces us to choose a sequential order for the calls to f (we evaluate $f(\mathsf{t})$ first, then $f(\mathsf{f})$). ■

3.2 Operational semantics

The substitution principle of Section 2.2, translated to the language of proof terms, says that for any substitution $\Gamma, \Gamma' \vdash \sigma : \Delta$ and arbitrary term $\Gamma, \Delta, \Gamma' \vdash t : J$ (i.e., a value $V : [P]$, function $F : P > Q$, substitution $\sigma' : \Delta'$, or expression $E : R$), there should be a term $[\sigma]t$ such that $\Gamma, \Gamma' \vdash [\sigma]t : J$. Rather than internalizing this principle in the syntax, we define $[\sigma]t$ as an operation, namely the usual simultaneous, capture-avoiding substitution. The definition

$\boxed{E \rightsquigarrow E'}$

$$\frac{\phi(p)\text{ defined}}{(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)} \quad \mathsf{id}(V) \rightsquigarrow V \quad \frac{E \rightsquigarrow E'}{F(E) \rightsquigarrow F(E')}$$

$\boxed{[\sigma]t}$

$$[\sigma]f = \begin{cases} F & (F/f) \in \sigma \\ f & f \notin \mathrm{dom}(\sigma) \end{cases}$$

$$[\sigma]([\sigma']p) = [[\sigma]\sigma']p \qquad [\sigma](\lambda\phi) = \lambda p.[\sigma]\phi(p) \qquad [\sigma]\mathsf{id} = \mathsf{id}$$

$$[\sigma]\cdot = \cdot \qquad [\sigma](F/f, \sigma') = ([\sigma]F/f, [\sigma]\sigma')$$

$$[\sigma](F(g(V))) = [\sigma]F([\sigma]g([\sigma]V))$$
$$[\sigma](F(V)) = [\sigma]F([\sigma]V) \qquad [\sigma](F(E)) = [\sigma]F([\sigma]E)$$

Figure 6. Focused λ-calculus (operational semantics)

of $[\sigma]t$ (given in Figure 6) is completely unsurprising, but a couple cases worth mention. Applying $[\sigma]$ to a function $\lambda\phi$ defines a new function by composing ϕ with the substitution:

$$[\sigma](\lambda\phi) = \lambda p.[\sigma]\phi(p)$$

Moreover, as we observed above, if σ maps G/g, then applying σ to the irreducible expression $F(g(V))$ converts it into two cuts: the expression $G([\sigma]V)$ composed with $[\sigma]F$.

The annotated version of the substitution principle is proven easily by induction, as in Section 2.2.

Lemma (Substitution)**.** *If* $\Gamma, \Gamma' \vdash \sigma : \Delta$ *and* $\Gamma, \Delta, \Gamma' \vdash t : J$ *and then* $\Gamma, \Gamma' \vdash [\sigma]t : J$.

The operational semantics is then given by a transition relation $E \rightsquigarrow E'$ on closed expressions, with three rules:

$$\frac{\phi(p)\text{ defined}}{(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)} \quad \mathsf{id}(V) \rightsquigarrow V \quad \frac{E \rightsquigarrow E'}{F(E) \rightsquigarrow F(E')}$$

All of the complexity of pattern-matching is implemented by the one rule on the left, so let's unpack it: a function $F = \lambda\phi$ is defined (syntactically) as a partial map from patterns to open expressions; a value $V = [\sigma]p$ is a pattern together with an explicit substitution for its variables; thus to *apply* F to V, we find the expression $\phi(p)$ corresponding to p (assuming one exists), and apply the substitution σ.

Preservation and progress are stated in the usual way.

Theorem (Preservation)**.** *If* $\Gamma \vdash E : P$ *and* $E \rightsquigarrow E'$*, then* $\Gamma \vdash E' : P$.

Proof. Immediate by induction on (the derivation of) $E \rightsquigarrow E'$, using the substitution lemma in the case of a reduction $(\lambda\phi)([\sigma]p) \rightsquigarrow [\sigma]\phi(p)$ (like in the proof of reduction from Section 2.2). □

Theorem (Progress)**.** *If* $\vdash E : P$*, then either* $E = V$ *or else there exists* E' *such that* $E \rightsquigarrow E'$.

Proof. Immediate by induction on $\vdash E : P$, using the fact that well-typed functions are exhaustive. □

EXAMPLE 3. Recall the functions and, not, and table1 from Examples 1 and 2. The reader can verify the following calculation:

$\mathsf{and}(\mathsf{table1}(\mathsf{not}))$
$\rightsquigarrow \mathsf{and}((\lambda b_1.(\lambda b_2.(b_1, b_2))(\mathsf{not}\ \mathsf{f}))(\mathsf{not}\ \mathsf{t}))$
$\rightsquigarrow \mathsf{and}((\lambda b_1.(\lambda b_2.(b_1, b_2))(\mathsf{not}\ \mathsf{f}))\ \mathsf{f})$
$\rightsquigarrow \mathsf{and}((\lambda b_2.(\mathsf{f}, b_2))(\mathsf{not}\ \mathsf{f}))$
$\rightsquigarrow \mathsf{and}((\lambda b_2.(\mathsf{f}, b_2))\ \mathsf{t})$
$\rightsquigarrow \mathsf{and}(\mathsf{f}, \mathsf{t})$
$\rightsquigarrow \mathsf{f}$

■

3.3 Representation in Coq

Yet the reader may still have lurking suspicions about our language definition. Aren't we overlooking Reynolds' lesson about the pitfalls of higher-order definitions of higher-order programming languages (Reynolds 1972)? Isn't there a circularity in our appeal to a "meta-level" notion of maps while defining functions? Here, we will attempt to rest these concerns by giving an encoding of focused λ-calculus in Coq, a proof assistant based on the Calculus of Inductive Constructions (Coquand and Huet 1988; Coquand and Paulin-Mohring 1989; Coq Development Team 2006).

But our first step will be to try to explain how in a paper with "higher-order abstract syntax" in the title, we will have the chutzpah to use de Bruijn indexes in this encoding (de Bruijn 1972). To be clear, we are proposing a new kind of higher-order abstract syntax. In its usual application, HOAS refers to representing object-language variables by meta-language variables (Pfenning and Elliott 1988). This allows object-language binding constructs to be encoded by corresponding meta-language constructs, and thereby eliminates the need for dealing explicitly with tricky notions such as variable-renaming, parametric quantification and capture-avoiding substitution. The logical framework Twelf is very well-suited for this kind of representation technique (Twelf 2007). In contrast, the novelty of our approach is encoding object-language *induction* by meta-level induction. The Coq proof assistant, it turns out, is well-suited for this kind of representation technique. Ideally, we would be able to combine both forms of HOAS, as we did above at a pre-formal level. But although there have been some attempts at encoding standard HOAS in Coq (Despeyroux et al. 1995), and some work on incorporating induction principles into LF (Schürmann et al. 2001), these are still at experimental stages. We therefore use Coq to highlight the novel aspects of our higher-order encoding, but accept the limitations of a first-order representation of variables.

With that apology out of the way, let us move on to the formalization.[4] As we did throughout the above discussion, we will define the focused λ-calculus in "Curry-style", that is, with typing rules for type-free terms, and a type-free operational semantics. An alternative "Church-style" approach would be to directly encode the logical rules of Section 2.2, and then simply extract the language, with *typed* terms being derivations of the logical judgments.[5]

We begin by defining $\mathsf{tp} : \mathsf{Set}$ as a standard algebraic datatype with constructors $0, 1 : \mathsf{tp}$ and $\otimes, \oplus, \xrightarrow{v} : \mathsf{tp} \to \mathsf{tp} \to \mathsf{tp}$ (we will use infix notation for the latter). For convenience, we also add $2 : \mathsf{tp}$ to directly represent booleans. The type of hypotheses $\mathsf{hyp} : \mathsf{Set}$ is defined by one constructor of type $\mathsf{tp} \to \mathsf{tp} \to \mathsf{hyp}$, but we will simply write $P \xrightarrow{v} Q : \mathsf{hyp}$, overloading the tp constructor (it will always be clear from context which constructor we really mean).

Now, linear contexts $\Delta : \mathsf{linctx}$ are lists of hyps, while stable contexts $\Gamma : \mathsf{ctx}$ are lists of linctxs. We write [] for the empty list, $[a]$ for a singleton, $a :: l$ for the "cons" operation, and $l_1 +\!+ l_2$ for concatenation. Since contexts are lists of *lists*, de Bruijn indexes are given by *pairs* of natural numbers, written $i.j : \mathsf{index}$. It is quite reasonable to think of these using machine intuitions: if Γ represents a stack of frames Δ, then a de Bruijn index $i.j$ specifies a "frame pointer" i plus an "offset" j. We write $\#j(\Delta)$ for the

[4] The full Coq source code for the encoding described here is available at: `http://www.cs.cmu.edu/~noam/research/focusing.tar`

[5] See `http://www.cs.cmu.edu/~noam/research/focus-church.v`.

jth element of Δ, and $\#i.j(\Gamma)$ for the jth element of the ith linear context in Γ. These are both partial operations, returning options in Coq, but we will abuse notation and write $\#i.j(\Gamma) = H$ meaning $\#i.j(\Gamma) = \mathsf{Some}\ H$, and similarly with $\#j(\Delta)$. In general, we will stray slightly from concrete Coq syntax so as to improve readability.

We define pat as another algebraic datatype, built using constructors $(), \mathsf{t}, \mathsf{f} : \mathsf{pat}$ and $(-,-) : \mathsf{pat} \to \mathsf{pat} \to \mathsf{pat}$, $\mathsf{inl}, \mathsf{inr} : \mathsf{pat} \to \mathsf{pat}$, and $\mathsf{fvar} : \mathsf{pat}$. The latter stands for a pattern binding a function variable—since we are using a de Bruijn representation, patterns do not actually name any variables. The pattern-typing judgment $\Delta \Rrightarrow p : P$ is encoded by an inductive type family $\mathsf{pat\_tp} : \mathsf{linctx} \to \mathsf{pat} \to \mathsf{tp} \to \mathsf{Prop}$. We omit the names of the constructors for pat_tp, but give their types below (also leaving implicit the $\forall$-quantification over all free variables):

$\_ : \mathsf{pat\_tp}\ [P \xrightarrow{v} Q]\ \mathsf{fvar}\ P \xrightarrow{v} Q$
$\_ : \mathsf{pat\_tp}\ []\ ()\ 1$
$\_ : \mathsf{pat\_tp}\ \Delta_1\ p_1\ P \to \mathsf{pat\_tp}\ \Delta_2\ p_2\ Q \to$
$\quad \mathsf{pat\_tp}\ (\Delta_1 +\!\!+ \Delta_2)\ (p_1, p_2)\ P \otimes Q$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ P \to \mathsf{pat\_tp}\ \Delta\ (\mathsf{inl}\ p)\ P \oplus Q$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ Q \to \mathsf{pat\_tp}\ \Delta\ (\mathsf{inr}\ p)\ P \oplus Q$

$\_ : \mathsf{pat\_tp}\ []\ \mathsf{t}\ 2$
$\_ : \mathsf{pat\_tp}\ []\ \mathsf{f}\ 2$

Now, the syntax of the language is defined through four mutually inductive types val, fnc, sub, and exp, with the following constructors:

$\mathsf{Value} : \mathsf{pat} \to \mathsf{sub} \to \mathsf{val}$

$\mathsf{Lam} : (\mathsf{pat} \to \mathsf{exp}) \to \mathsf{fnc}$
$\mathsf{Id} : \mathsf{fnc}$
$\mathsf{IdVar} : \mathsf{index} \to \mathsf{fnc}$

$\mathsf{Subst} : \mathsf{list\ fnc} \to \mathsf{sub}$

$\mathsf{Return} : \mathsf{val} \to \mathsf{exp}$
$\mathsf{Comp} : \mathsf{fnc} \to \mathsf{index} \to \mathsf{val} \to \mathsf{exp}$
$\mathsf{AppV} : \mathsf{fnc} \to \mathsf{val} \to \mathsf{exp}$
$\mathsf{AppE} : \mathsf{fnc} \to \mathsf{exp} \to \mathsf{exp}$
$\mathsf{Fail} : \mathsf{exp}$

As promised, fnc contains maps from patterns to expressions, embedded through the constructor $\mathsf{Lam} : (\mathsf{pat} \to \mathsf{exp}) \to \mathsf{fnc}$. Note that this is a positive definition (and thus acceptable in Coq) because the type pat was already defined—as opposed to, say, the definition $\mathsf{Lam}' : (\mathsf{val} \to \mathsf{exp}) \to \mathsf{fnc}$ (which would be illegal in Coq). On the other hand, Coq requires maps $\mathsf{pat} \to \mathsf{exp}$ to be *total,* so to simulate partial maps we add an expression $\mathsf{Fail} : \mathsf{exp}$, which can be read as "undefined" or "stuck".

Following the representation of linear contexts as unlabelled lists of hypotheses, a substitution is just an unlabelled list of functions, while the expression $\mathsf{Return}\ V$ makes explicit the implicit inclusion of values into expressions. Otherwise, the constructors are all straightforward transcriptions of terms of focused λ-calculus.

In the following examples, we abbreviate $\mathsf{Value}\ p\ (\mathsf{Subst}\ [])$ by $\ulcorner p \urcorner$, and $\mathsf{Return}\ (\ulcorner p \urcorner)$ by $\lceil p \rceil$.

EXAMPLE 4. The Coq encodings of and, not : fnc are:

$$\mathsf{and} = \mathsf{Lam} \begin{pmatrix} (\mathsf{t},\mathsf{t}) & \mapsto & \lceil \mathsf{t} \rceil \\ (\mathsf{t},\mathsf{f}) & \mapsto & \lceil \mathsf{f} \rceil \\ (\mathsf{f},\mathsf{t}) & \mapsto & \lceil \mathsf{f} \rceil \\ (\mathsf{f},\mathsf{f}) & \mapsto & \lceil \mathsf{f} \rceil \\ \_ & \mapsto & \mathsf{Fail} \end{pmatrix}$$

$$\mathsf{not} = \mathsf{Lam}(\mathsf{t} \mapsto \lceil \mathsf{f} \rceil \mid \mathsf{f} \mapsto \lceil \mathsf{t} \rceil \mid \_ \mapsto \mathsf{Fail})$$

These definitions make use of Coq's built-in pattern-matching facilities to in order to pattern-match on pats. ■

EXAMPLE 5. The encoding of $\mathsf{table1} : \mathsf{fnc}$ makes careful use of de Bruijn indexes:

$$\mathsf{table1} = \mathsf{Lam} \begin{pmatrix} \mathsf{fvar} \mapsto \mathsf{Comp}\ (\mathsf{Lam}(b_1 \mapsto \\ \qquad \mathsf{Comp}\ (\mathsf{Lam}(b_2 \mapsto \lceil (b_1, b_2) \rceil)) \\ \qquad 1.0\ \ulcorner \mathsf{f} \urcorner))\ 0.0\ \ulcorner \mathsf{t} \urcorner \\ \_ \quad \mapsto \mathsf{Fail} \end{pmatrix}$$

In the first call (with value t), the function argument is (the first and only entry) on the top of the stack, so we reference it by 0.0. In the second call, a frame (coincidentally empty) has been pushed in front of the function, so we reference it by 1.0. ■

Now we build the four typing-judgments as mutually inductive type-families, defined as follows (again omitting constructors for the typing rules, and outermost $\forall$-quantifiers):

$\mathsf{val\_tp} : \mathsf{ctx} \to \mathsf{tp} \to \mathsf{Prop}$
$\_ : \mathsf{pat\_tp}\ \Delta\ p\ P \to \mathsf{sub\_tp}\ \Gamma\ \sigma\ \Delta \to \mathsf{val\_tp}\ \Gamma\ (\mathsf{Value}\ p\ \sigma)\ P$

$\mathsf{fnc\_tp} : \mathsf{ctx} \to \mathsf{tp} \to \mathsf{tp} \to \mathsf{Prop}$
$\_ : (\forall p \forall \Delta.\mathsf{pat\_tp}\ \Delta\ p\ P \to \mathsf{exp\_tp}\ (\Delta :: \Gamma)\ \phi(p)\ Q)$
$\quad \to \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{Lam}\ \phi)\ P\ Q$
$\_ : \mathsf{fnc\_tp}\ \mathsf{Id}\ P\ P$
$\_ : (\#i.j(\Gamma) = (P \xrightarrow{v} Q)) \to \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{IdVar}\ i.j)\ P\ Q$

$\mathsf{sub\_tp} : \mathsf{ctx} \to \mathsf{linctx} \to \mathsf{Prop}$
$\_ : \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ [])\ []$
$\_ : \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \to \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ \sigma)\ \Delta$
$\quad \to \mathsf{sub\_tp}\ \Gamma\ (\mathsf{Subst}\ (F :: \sigma))\ (P \xrightarrow{v} Q :: \Delta)$

$\mathsf{exp\_tp} : \mathsf{ctx} \to \mathsf{tp} \to \mathsf{Prop}$
$\_ : \mathsf{val\_tp}\ \Gamma\ V\ P \to \mathsf{exp\_tp}\ \Gamma\ (\mathsf{Return}\ V)\ P$
$\_ : (\#i.j(\Gamma) = (P \xrightarrow{v} Q)) \to \mathsf{val\_tp}\ \Gamma\ V\ P \to \mathsf{fnc\_tp}\ \Gamma\ F\ Q\ R$
$\quad \to \mathsf{exp\_tp}\ \Gamma\ (\mathsf{Comp}\ F\ i.j\ V)\ R$
$\_ : \mathsf{val\_tp}\ \Gamma\ V\ P \to \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \to \mathsf{exp\_tp}\ \Gamma\ (\mathsf{AppV}\ F\ V)\ Q$
$\_ : \mathsf{exp\_tp}\ \Gamma\ E\ P \to \mathsf{fnc\_tp}\ \Gamma\ F\ P\ Q \to \mathsf{exp\_tp}\ \Gamma\ (\mathsf{AppE}\ F\ E)\ Q$

Again, these definitions are a direct transcription of the typing rules in Figure 5, including the higher-order rule for function-typing.

Finally, to encode the operational semantics, we first define the different substitution operations:

$\mathsf{sub\_val} : \mathsf{nat} \to \mathsf{sub} \to \mathsf{val} \to \mathsf{val}$
$\mathsf{sub\_fnc} : \mathsf{nat} \to \mathsf{sub} \to \mathsf{fnc} \to \mathsf{fnc}$
$\mathsf{sub\_sub} : \mathsf{nat} \to \mathsf{sub} \to \mathsf{sub} \to \mathsf{sub}$
$\mathsf{sub\_exp} : \mathsf{nat} \to \mathsf{sub} \to \mathsf{exp} \to \mathsf{exp}$

These are defined by (mutual) structural induction on the term being substituted into, essentially as in Figure 6, but with a bit of extra reasoning about de Bruijn indices. The extra nat argument is a frame pointer to the linear context Δ being substituted for, and is used as follows in the IdVar case (and analogously in the Comp case):

$$\mathsf{sub\_fnc}\ i\ \sigma\ (\mathsf{IdVar}\ i'.j) = \begin{cases} \#j(\sigma) & i = i' \\ \mathsf{IdVar}\ i'.j & i > i' \\ \mathsf{IdVar}\ (i'-1).j & i < i' \end{cases}$$

We can then define the transition relation as an inductive family $\mathsf{step} : \mathsf{exp} \to \mathsf{exp} \to \mathsf{Prop}$, with the following rules:

$\_ : \mathsf{step}\ (\mathsf{AppV}\ (\mathsf{Lam}\ \phi)\ (\mathsf{Value}\ p\ \sigma))\ (\mathsf{sub\_exp}\ 0\ \sigma\ \phi(p))$
$\_ : \mathsf{step}\ (\mathsf{AppV}\ \mathsf{Id}\ V)\ (\mathsf{Return}\ V)$
$\_ : \mathsf{step}\ E\ E' \to \mathsf{step}\ (\mathsf{AppE}\ F\ E)\ (\mathsf{AppE}\ F\ E')$
$\_ : \mathsf{step}\ (\mathsf{AppE}\ F\ (\mathsf{Return}\ V))\ (\mathsf{AppV}\ F\ V)$

These mirror the rules in Figure 6, with one additional rule for the (formerly implicit) transition from composition to reduction after the expression argument has been reduced to a value.

Finally, we define a predicate $\mathsf{terminal} : \mathsf{exp} \to \mathsf{Prop}$ and assert $\_ : \mathsf{terminal}\ (\mathsf{Return}\ V)$. Given these definitions, we can state the preservation and progress theorems:

$$\mathsf{preservation} : \mathsf{exp\_tp}\ \Gamma\ E\ P \to \mathsf{step}\ E\ E' \to \mathsf{exp\_tp}\ \Gamma\ E'\ P$$
$$\mathsf{progress} : \mathsf{exp\_tp}\ []\ E\ P \to (\mathsf{terminal}\ E \vee \exists E'.\mathsf{step}\ E\ E')$$

Both theorems have short proofs in Coq, constructed using the tactic language. As in the paper proof, the preservation theorem relies on the substitution principle, which in turn relies on weakening. Both substitution and weakening require establishing a few trivial facts about arithmetic, lists, and de Bruijn indices. This (about 140 lines to prove the trivial lemmas, followed by about 230 lines to prove weakening and substitution, much of it dealing simply with the coding of mutual induction principles in Coq) is the main source of bureaucracy in the Coq formalization, which otherwise follows our informal presentation very closely.

3.4 Recursion and recursive types

We have seen how focusing the λ-calculus gives logical explanations for notions such as pattern-matching and evaluation order, which are typically seen as extra-logical. Once we have this analysis, we can extend the language in a fairly open-ended way without modifying the logical core. In this section, we will consider two particularly easy extensions: recursion and recursive types. For recursive functions, we add one typing rule and one evaluation rule:

$$\frac{\Gamma, f : P \xrightarrow{v} Q \vdash F : P > Q}{\Gamma \vdash \mathsf{fix}\ f.F : P > Q} \qquad (\mathsf{fix}\ f.F)(V) \rightsquigarrow ([\mathsf{fix}\ f.F/f]F)(V)$$

These rules can be transcribed directly (modulo de Bruijn indices) into Coq:

$$\_ : \mathsf{fnc\_tp}\ (\Gamma, [P \xrightarrow{v} Q])\ F\ P\ Q \to \mathsf{fnc\_tp}\ \Gamma\ (\mathsf{fix}\ F)\ P\ Q$$
$$\_ : \mathsf{step}\ (\mathsf{AppV}\ (\mathsf{fix}\ F)\ V)\ (\mathsf{AppV}\ (\mathsf{sub\_fnc}\ 0\ (\mathsf{Subst}\ [\mathsf{fix}\ F])\ F)\ V)$$

To verify the safety of this extension, we need only localized checks: one extra case each in the proofs of weakening, substitution, preservation, and progress.

EXAMPLE 6. We define $\mathsf{loop} : 1 > 1 = \mathsf{fix}\ f.\lambda().f()$. Then $\mathsf{loop}() \rightsquigarrow (\lambda().\mathsf{loop}())() \rightsquigarrow \mathsf{loop}() \rightsquigarrow \ldots$. ■

For recursive types, we add a single *pattern*-typing rule:

$$\frac{\Delta \Rightarrow p : [\mu X.P/X]P}{\Delta \Rightarrow \mathsf{fold}(p) : \mu X.P}$$

To add general μ-types to our Coq formalization, we would have to introduce the additional bureaucracy of type substitution. On the other hand, for particular recursive types (such as those considered below) we can directly transcribe their pattern-typing rules. And these rules suffice: we do not have to extend or modify any other aspect of the type system or operational semantics. The machinery of focusing and higher-order abstract syntax gives us the value-forming rules and pattern-matching on recursive types "for free". In particular, our proof of type safety (both on paper and in the Coq formalization) needs absolutely no modification, since it references the pattern-typing judgment uniformly.

EXAMPLE 7. In this example, we consider natural numbers $\mathsf{Nat} = \mu X.1 \oplus X$, defined by two pattern-typing rules:

$$\frac{}{\cdot \Rightarrow \mathsf{Z} : \mathsf{Nat}} \qquad \frac{\cdot \Rightarrow p : \mathsf{Nat}}{\cdot \Rightarrow \mathsf{S}\ p : \mathsf{Nat}}$$

We can encode the plus function like so:

$$\mathsf{plus} = \mathsf{fix}\ f.\lambda \begin{pmatrix} (m, \mathsf{Z}) & \mapsto & m \\ (m, \mathsf{S}\ n) & \mapsto & (\lambda n'.\mathsf{S}\ n') f(m, n) \end{pmatrix}$$

For instance, $\mathsf{plus}\ (\mathsf{S}(\mathsf{S}\ \mathsf{Z}), \mathsf{S}\ \mathsf{Z}) \rightsquigarrow^* \mathsf{S}(\mathsf{S}(\mathsf{S}\ \mathsf{Z}))$. To verify that $\mathsf{plus} : \mathsf{Nat} \otimes \mathsf{Nat} > \mathsf{Nat}$, we must check that for any $\mathsf{Nat} \otimes \mathsf{Nat}$ pattern, there is a corresponding Nat-typed branch of the function. This is easily seen to be true, since all $\mathsf{Nat} \otimes \mathsf{Nat}$ patterns have the form (m, Z) or $(m, \mathsf{S}\ n)$. ■

EXAMPLE 8. Consider a domain $\mathsf{D} = \mu X.1 \oplus \mathsf{Nat} \oplus (X \xrightarrow{v} X)$:

$$\frac{}{\cdot \Rightarrow \mathsf{U} : \mathsf{D}} \qquad \frac{\cdot \Rightarrow p : \mathsf{Nat}}{\cdot \Rightarrow \mathsf{N}\ p : \mathsf{D}} \qquad \frac{}{f : \mathsf{D} \xrightarrow{v} \mathsf{D} \Rightarrow \mathsf{F}\ f : \mathsf{D}}$$

We define a function $\mathsf{app} : \mathsf{D} \otimes \mathsf{D} > \mathsf{D}$, which tries to apply the first argument to the second (and returns U if the first argument is not a function):

$$\mathsf{app} = \lambda \begin{pmatrix} (F\ f, d) & \mapsto & \mathsf{id}(f(d)) \\ (\_, d) & \mapsto & \mathsf{U} \end{pmatrix}$$

For instance, $\mathsf{app}\ (\mathsf{F}\ \mathsf{id}, V) \rightsquigarrow \mathsf{id}(\mathsf{id}(V)) \rightsquigarrow \mathsf{id}(V) \rightsquigarrow V$. ■

While these examples illustrate the simplicity of higher-order syntax for pattern-matching on recursive types, they also raise some subtle theoretical questions. A careful reader might have noticed that there is another way of defining the plus function in Coq: rather than explicitly using the fix operator, we could use Coq's built-in Fixpoint mechanism to define a map $\mathsf{plus\_pat} : \mathsf{pat} \to \mathsf{pat} \to \mathsf{pat}$ computing the sum of two Nat patterns, and then define

$$\mathsf{plus}^* : \mathsf{fnc} = \mathsf{Lam}((m, n) \mapsto \lceil \mathsf{plus\_pat}\ m\ n \rceil \mid \_ \mapsto \mathsf{Fail})$$

Strictly speaking, plus^* is an "exotic term", i.e., does not represent a term of concrete syntax (Despeyroux et al. 1995), since it corresponds to a function defined by infinitely many pattern-branches. Operationally, it computes the sum of two numbers in a single step of evaluation, whereas plus computes it in multiple steps (linear in n). Nonetheless, plus and plus^* are observationally equivalent. We conjecture that this is always the case, and that any term definable in the Coq encoding of focused λ-calculus with recursive types is observationally equivalent to a term of concrete syntax using explicit recursion. Yet, even if this conjecture holds, it is an interesting question whether there is a principled way to adapt the HOAS encoding to eliminate terms such as plus^* altogether.

Proof-theoretically speaking, we can put it this way: for some recursive types P, establishing $\Gamma; P \vdash Q$ requires an infinitely wide derivation in the focusing system.[6] Conversely, for other recursive types, the identity principle $\Gamma; P \vdash P$ requires a derivation that is infinitely *deep*. In particular, the subformula relationship may not be well-founded (e.g., $\mathsf{D} \xrightarrow{v} \mathsf{D}$ is a subformula of D). This is not really an issue for our programming language: rather than attempting an infinite derivation, we can simply invoke the internalized identity principle. More fundamentally, though, we can give these derivations a coinductive reading—this becomes particularly significant if we want to extend the language and incorporate *subtyping* through an identity-coercion interpretation, as explored by Brandt and Henglein (1998).

[6] E.g., for Nat we essentially have the ω-rule (Buchholz et al. 1981).

4. Related work

This paper is by no means the first to propose a logical explanation for pattern-matching or explicit substitutions. Recently, Nanevski et al. (2007) and Pientka (2008) offer a judgmental explanation for explicit substitutions in a modal type theory. Methodologically their work is close to ours, but their development is rather different since they seek to understand the connection between explicit substitutions and *meta-variables* (as used, e.g., in logical frameworks and staged computation), rather than pattern-matching. Cerrito and Kesner (2004) give an interpretation for both nested patterns and explicit substitutions in sequent calculus. It seems the difficulty with taking the unfocused sequent calculus as a starting point, though, is that it suffers from a "lack of judgments"—to explain pattern-matching Cerrito and Kesner must introduce additional scaffolding beyond the Curry-Howard isomorphism. For example, to obtain a well-behaved language with substitution and subject reduction, they must annotate the single cut rule of sequent calculus as three different typing rules, and add another typing rule (*app*) with little proof-theoretic motivation. In contrast, every typing rule we gave in Section 3.1 was either a direct annotation of a logical rule in Section 2.2, or else internalized one of the cut or identity principles.

An additional byproduct of our use of focusing as a logical foundation is that the extracted language has explicit evaluation order. As mentioned in the Introduction, this connection has been explored before by various people, at first with only a loose tie to linear logic (Curien and Herbelin 2000; Selinger 2001; Wadler 2003), but later with an explicit appeal to polarity and focusing (Laurent 2005; Dyckhoff and Lengrand 2006). From this line of work, our main technical innovation is the uniform treatment of the positive connectives through pattern-matching, which considerably simplifies previous formalisms while allowing us to consider a rich set of connectives. Our approach is loosely inspired by that of Girard (2001).

In a short but prescient paper, Coquand (1992) examines pattern-matching as an alternative to the usual elimination rules in the framework of Martin-Löf's type theory, and concludes with an offhand remark, "From a proof-theoretic viewpoint, our treatment can be characterized as fixing the meaning of a logical constant by its introduction rules". We have seen how this interpretation arises naturally out of focusing for the positive connectives, although how to extend our approach to the dependently-typed case remains an important open question. Elsewhere, we explore the dual interpretation for the negative connectives (and lazy evaluation), tying the unified analysis to Michael Dummett's examination of the justification of logical laws (Dummett 1991; Zeilberger 2007).

Acknowledgments

Special thanks to Frank Pfenning for his invaluable guidance during the development of this work, and for suggestions on improving its presentation. I would also like to thank Bob Harper, Neel Krishnaswami, Peter Lee, William Lovas, Dan Licata, Jason Reed, Rob Simmons, and other members of the CMU PL group for lively discussions, and the POPL reviewers for their helpful comments. Finally, I am grateful to the `coq-club` mailing list for useful advice on Coq, and particularly to Xavier Leroy for explaining a cool trick for coding mutual induction principles.

References

M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, 1991.

Jean-Marc Andreoli. Focussing and proof construction. *Annals of Pure and Applied Logic*, 107(1):131–163, 2001.

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

Michael Brandt and Fritz Henglein. Coinductive axiomatization of recursive type equaility and subtyping. *Fundamenta Informaticae*, 20:1–24, 1998.

W. Buchholz, S. Feferman, W. Pohlers, and W. Sieg. *Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies*. Springer-Verlag, 1981.

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

The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 8.1*. INRIA, 2006. `http://coq.inria.fr/doc/main.html`.

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

Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2/3):95–120, 1988.

Thierry Coquand and Christine Paulin-Mohring. Inductively defined types. In *LNCS 389*. Springer-Verlag, 1989.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ICFP '00: Proceedings of the SIGPLAN International Conference on Functional Programming*, pages 233–243. 2000.

Nicolaas G. de Bruijn. A lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, volume 902 of *LNCS*, pages 124–138, Edinburgh, Scotland, 1995. Springer-Verlag.

Michael Dummett. *The Logical Basis of Metaphysics*. The William James Lectures, 1976. Harvard University Press, Cambridge, Massachusetts, 1991. ISBN 0-674-53785-8.

Roy Dyckhoff and Stephane Lengrand. LJQ: A strongly focused calculus for intuitionistic logic. In *Proceedings of the Second Conference on Computability in Europe*, 2006.

Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

Gerhard Gentzen. Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39:176–210, 405–431, 1935. English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, pages 68–131, North-Holland, 1969.

Jean-Yves Girard. Locus solum: From the rules of logic to the logic of rules. *Mathematical Structures in Computer Science*, 11(3):301–506, 2001.

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

Hugo Herbelin. A lambda-calculus structure isomorphic to Gentzen-style sequent calculus structure. In *CSL '94: Proceedings of the 8th International Workshop on Computer Science Logic*, 1995.

Jacob M. Howe. *Proof search issues in some non-classical logics*. PhD thesis, University of St Andrews, December 1998. URL `http://www.cs.kent.ac.uk/pubs/1998/946`. Available as University of St Andrews Research Report CS/99/1.

Steven C. Kleene. *Introduction to Metamathematics*. Van Nostrand, Princeton, NJ, 1952.

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

Olivier Laurent. Classical isomorphisms of types. *Mathematical Structures in Computer Science*, 15(5):969–1004, October 2005.

Per 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.

Per Martin-Löf. On the meanings of the logical constants and the justifications of the logical laws. *Nordic Journal of Philosophical Logic*, 1(1):11–60, 1996. URL `http://www.hf.uio.no/filosofi/njpl/vol1no1/meaning/meaning.html`.

Dale Miller and Chuck Liang. Focusing and polarization in intuitionistic logic. In *CSL '07: Proceedings of the 21st International Workshop on Computer Science Logic*. 2007.

Eugenio Moggi. Notions of computation and monads. *Information and Compution*, 93(1):55–92, 1991.

Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *Transactions on Computational Logic*, 2007. To appear.

Frank Pfenning and Rowan Davies. A judgmental reconstruction of modal logic. *Mathematical Structures in Computer Science*, 11(4):511–540, 2001.

Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *PLDI '88: Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, 1988.

Brigitte Pientka. A type-theoretic foundation for programming with higher-order abstract syntax and first-class substitutions. In *POPL '08: Proceedings of the SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM '72: Proceedings of the ACM annual conference*, pages 717–740, 1972.

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

Peter Selinger. Control categories and duality: on the categorical semantics of the lambda-mu calculus. *Mathematical Structures in Computer Science*, 11(2):207–260, 2001.

Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*, volume Cambridge Tracts in Theoretical Computer Science 43. Cambridge University Press, 1996.

Twelf 2007. *The Twelf Project*, 2007. `http://twelf.plparty.org/`.

Philip Wadler. Call-by-value is dual to call-by-name. In *ICFP '03: Proceedings of the SIGPLAN International Conference on Functional Programming*, pages 189–201, 2003.

Philip Wadler. A taste of linear logic. In *MFCS: Symposium on Mathematical Foundations of Computer Science*, 1993.

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