Defunctionalizing Focusing Proofs

(Or, How Twelf Learned To Stop Worrying And Love The Ω-rule)

Noam Zeilberger
noam@cs.cmu.edu

Carnegie Mellon University

Abstract. In previous work, the author gave a higher-order analysis of *focusing proofs* (in the sense of Andreoli's search strategy), with a role for infinitary rules very similar in structure to Buchholz's Ω-rule. Among other benefits, this "pattern-based" description of focusing simplifies the cut-elimination procedure, allowing cuts to be eliminated in a connective-generic way. However, interpreted literally, it is problematic as a representation technique for proofs, because of the difficulty of inspecting and/or exhaustively searching over these infinite objects. In the spirit of infinitary proof theory, this paper explores a view of pattern-based focusing proofs as *façons de parler,* describing how to compile them down to first-order derivations through defunctionalization, Reynolds' program transformation. Our main result is a representation of pattern-based focusing in the Twelf logical framework, whose core type theory is too weak to directly encode infinitary rules—although this weakness directly enables so-called "higher-order abstract syntax" encodings. By applying the systematic defunctionalization transform, not only do we retain the benefits of the higher-order focusing analysis, but we can also take advantage of HOAS within Twelf, ultimately arriving at a proof representation with surprisingly little bureaucracy.

1 Introduction

Part of Hilbert's program was to prove the consistency of all mathematics using only finitary methods. Gödel showed this to be impossible, but nonetheless the program carried on, relaxing the finitary restriction. A seminal achievement was Gentzen's proof of the consistency of arithmetic through the technique of cut-elimination [1]. Gentzen defined a finitary sequent calculus for Peano arithmetic, and then described an entirely finitary procedure for reducing cuts in these proofs, driving them towards a normal form which self-evidently excludes proofs of contradiction. However, the argument that the procedure terminates relied on a transfinite induction.

Schütte [2] observed that Gentzen's consistency proof could be considerably simplified by applying infinitary arguments *within* the proof system, not just when reasoning about it meta-theoretically. Specifically, he considered translating Gentzen's sequent calculus with the standard rule of arithmetic induction into a sequent calculus where the induction rule was replaced by the so-called ω-rule (proposed by Hilbert [3], studied also by Novikov [4] and Lorenzen [5]):

$$\frac{\Gamma \vdash A(0) \quad \Gamma \vdash A(1) \quad \dots \quad \Gamma \vdash A(n) \quad \dots}{\Gamma \vdash \forall x.A(x)}$$

The ω-rule has infinitely many premises, deriving a universal statement $\forall x.A(x)$ given proofs of $A(n)$ for each n. At face, this may seem like an absurd rule of inference—how can we ever get our hands on the infinitely many proofs required to deduce $\forall x.A(x)$? But the first thing to keep in mind when reading the rule is that the ellipses need not be taken literally, but more as a "way of speaking". Instead of seeing infinitely many premises in the ω-rule, we can view the rule as demanding a symbolic *function* from numbers n to proofs of $A(n)$. Such an interpretation was considered for example by Shoenfield [6]. The second thing to keep in mind about the ω-rule is that it offers a real simplification over the induction rule when analyzing a sequent calculus for arithmetic—specifically, one can reduce to proofs that are genuinely cut-free, rather than to proofs with cuts of a restricted form.

Several decades after Schütte's work, Buchholz [7] further demonstrated the power of infinitary rules for analyzing finite proof systems (in this case for iterated inductive definitions). Buchholz invented a new infinitary rule, the Ω-rule (capitalization important!), which can be displayed as follows:

$$\frac{\Delta \Vdash_u P(n) \quad \longrightarrow \quad \Gamma, \Delta \vdash_v C}{\Gamma \vdash_v P(n) \supset C} \; \Omega_{u+1}$$

Here we have two ordinals $u \prec v$ and some arithmetic proposition $P(n)$, and we assume that the natural deduction system ID_u^∞ has already been fully defined. The Ω_{u+1}-rule is then included in the definition of a new system ID_v^∞. Intuitively, the rule says that we can reason from $P(n)$ towards C in ID_v^∞, if we can convert any "direct" proof of $P(n)$ in ID_u^∞ into a proof of C in ID_v^∞. More precisely, the premise requires that we can convert any *normal* deduction (in the sense of Prawitz [8]) of $P(n)$ with assumptions Δ into a proof of C under the additional assumptions Δ. There is an additional technical restriction that the assumptions Δ consist only of *negative* formulas, in a sense similar to Girard's polarity [9].

In earlier work, I described a connection between something like Buchholz's Ω-rule and Andreoli's *focusing proofs.* For the purposes of this paper, I will call this analysis *pattern-based focusing.*[1] In Andreoli's original presentation [16], focusing proofs can be seen as defining a complete search strategy for sequent calculus, alternating between dual "inversion" and "focus" phases on formulas in the left or right side of a sequent. In pattern-based focusing, the left-inversion phase (for arbitrary positive formulas) is described as an instance of the Ω-rule, and the right-focus phase (again for arbitrary positive formulas) is revealed as a natural dual of the Ω-rule: to prove a proposition, find a direct proof of it under assumptions Δ in the more primitive system, and satisfy all the assumptions Δ in the current system. Both rules also have formal duals, corresponding to right-inversion and left-focus for negative formulas. For example, the right-inversion phase can be described with an Ω-like rule that quantifies over direct refutations rather than direct proofs.

Rather than an iterated hierarchy of proof systems, pattern-based focusing really only uses a single iteration of the Ω-like rules, from a base system to a second-order derived system. The base system is taken literally as a *definition* of the logical connectives. Through the Ω-rules and their duals, this definition is then used to derive appropriate rules of inference for the connectives in the second-order system. Proof-theoretically, the benefit of pattern-based focusing over the traditional presentation is that it improves modularity—for example, the cut-elimination procedure can be described in a connective-generic way. It also has an interesting operational interpretation through the proofs-as-programs correspondence. For example, the Ω-rule for right-inversion corresponds to a rule for building call-by-value continuations by pattern-matching: to define a continuation for (a positive type) A, we must build a map from A-patterns to computations. The cut-elimination procedure corresponds directly to a simple operational semantics for evaluating pattern-matching programs with deterministic evaluation order.

Despite these advantages, the pattern-based focusing analysis has the same trouble as the original ω-rule in being difficult to make sense of directly. As we did with the ω-rule, we can interpret these Ω-like rules constructively, replacing infinitely many premises with a single symbolic function. But this is still somewhat nebulous—just what are these "symbolic functions"? Nothing in the abstract description of the proof system forces us into adopting one particular interpretation, so in order to answer this question we should first ask just what we are hoping to *do* with proofs. From the point of view of proof-checking, it is important that we be able to inspect these functions *intensionally,* because their domain is potentially infinite (or rather, because it is potentially non-compact [17]), in which case we could never verify extensionally that a proof really is a proof. From the point of view of proof-search, it is important that we be able to *enumerate* these functions. From the point of view of extracting computational content from proofs, neither of the above properties is necessarily important, and we might only require that functions be computable (and even that could be relaxed, opting for an oracular operational semantics).

This paper is a preliminary exploration in representations of proofs for these different domains, starting from the pattern-based analysis of focusing. Our main weapon will be an old technique due to Reynolds called

[1] This analysis was described in [10] and refined/extended in some later papers [11,12,13,14]. The presentation here is modeled more closely on Chapters 2 and 3 of my thesis [15], which improves the notation of [10], and corrects some of its philosophical confusions.

defunctionalization [18], which systematically transforms programs with higher-order functions into first-order programs. In the traditional setting of functional programming, the insight behind defunctionalization is that a given program mentions only finitely many symbolic functions (a.k.a. λ-expressions), so that each can be assigned a unique *tag*. Then an *apply* function defined over the set of tags explains how to invoke the function denoted by a tag to any appropriate argument. So the defunctionalization transformation simply consists of two easy steps: replace each function abstraction in the program by the corresponding tag, and replace each function application by a call to "apply". The resulting program contains only a single, first-order function (or a collection of first-order functions, if we distinguish apply functions by type). On the other hand, we can still see the original higher-order structure implicit in the defunctionalized program—formally, defunctionalization has a left inverse, *refunctionalization* [19,20].

Some previous papers described type-theoretic representations of pattern-based focusing in Coq [11] and Agda [12], where the Ω-rules were interpreted literally as higher-order constructors. These frameworks are amenable to such literal interpretations because their core type theories (descended from Martin-Löf's "theory of sets" [21]) include the full power of iterated inductive definitions. In contrast, the Twelf logical framework [22] does not allow such encodings, because its core type theory, LF [23] (descended from Martin-Löf's "system of arities"), includes only very limited forms of functions: those which do not inspect their arguments. The weakness of the LF function space is also a strength, because it directly enables so-called "higher-order abstract syntax" (HOAS) encodings [24], avoiding the bureaucracy typically necessary for dealing with variable-binding and substitution.[2]

The main result of this paper is an encoding, via defunctionalization, of pattern-based focusing in Twelf. As speculated in [11], the benefit of this representation is that we are able to use *both* techniques: by making the higher-order rules of pattern-based focusing implicit through the use of defunctionalization, while taking explicit advantage of LF-style higher-order abstract syntax, we arrive at a concrete, formal representation of proofs with surprisingly little bureaucracy.

The first half of this paper gives a review of the pattern-based focusing analysis (modeled on [15, Ch. 2]), and the second half describes the Twelf encoding and the defunctionalization transformation. Some familiarity with Twelf is probably necessary for understanding the second half, although the ideas can be transported to other languages built on LF, such as Beluga [25] or Delphin [26]. Note that the idea of defunctionalizing focusing proofs was described cursorily in [14] for the purpose of building a refinement type-checker in Agda, but here we pursue it in greater depth, while taking advantage of the peculiar capabilities of Twelf.

2 Patterns and canonical derivations

Definition 1 (Polarized propositions). *We use the letters A, B, C to range over propositions. Every proposition has a definite* **polarity***, positive or negative. We indicate the polarity of a proposition A with a superscript A^+ or A^-, unless it is unimportant or already clear from context.*

Definition 2 (Affirmation/Denial). *We write "A true" for the* **affirmation** *of A, abbreviated "A", and "A false" for the* **denial** *of A, abbreviated "$\bullet A$".*

Definition 3 (Frames and simple frames). *A* **frame** *Δ is a tree of affirmations and denials. We write $\cdot$ for the empty frame, and Δ_1, Δ_2 for the join of two frames. We write $\Delta \in \Delta'$ if Δ occurs as a subtree of Δ'. We call $B \in \Delta$ and $\bullet B \in \Delta$ the* **holes** *of Δ. A frame without any holes B^+ or $\bullet B^-$ (i.e., with only positive denials $\bullet B^+$ and negative affirmations B^-) is called* **simple**.[3]

Definition 4 (Dictionaries of patterns, definition ordering). *A* **dictionary** *Ψ is an inductively defined relation $\Delta \Vdash \Delta'$, where Δ is a simple frame and Δ' is either A^+ or $\bullet A^-$. Derivations of $\Delta \Vdash A^+$ are called* **proof patterns***, and derivations of $\Delta \Vdash \bullet A^-$ are called* **refutation patterns***. In these patterns, we refer*

[2] The term "higher-order abstract syntax" was overloaded in [11] to stand for the use of Ω-rules to represent pattern-matching, but obviously these are two very different representation techniques: a function that does not inspect its argument is the most degenerate case of a pattern-matching function.

[3] For simplicity (so to speak), we omit atoms in this paper. In general, a simple frame can have holes X^+ and $\bullet X^-$.

to the frame Δ as the "frame of" the pattern. Given a dictionary, we describe the **definition ordering** $\prec$ *as the least transitive relation satisfying the following:*

- *If $\Delta \Vdash A^+$ then $\Delta \prec A^+$, and if $\Delta \Vdash \bullet A^-$ then $\Delta \prec A^-$*
- *If $\bullet B^+ \in \Delta$ then $B^+ \prec \Delta$, and if $B^- \in \Delta$ then $B^- \prec \Delta$*

For arbitrary propositions A, we write $\prec_A$ for the restriction of $\prec$ below A, and similarly $\prec_\Delta$ for the restriction of $\prec$ below Δ.

The intuition behind patterns is that they describe the shape of logical derivations, up to some holes for proofs (of negative propositions) and refutations (of positive propositions). The key idea is that positive propositions are literally *defined* by their proof patterns, and negative propositions by their refutation patterns, and hence the collection of all patterns is a "dictionary" for interpreting propositions. The definition ordering induced by a dictionary is a more abstract notion of *subformula* (cf. [27]). The ordering need not be well-founded, although this will be the case for propositions built out of the standard (polarized) connectives.

Example 1. The dictionary Ψ_{pol} contains the following inductive clauses:

$$\frac{}{\cdot \Vdash 1} \qquad \frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash B^+}{\Delta_1, \Delta_2 \Vdash A^+ \otimes B^+} \qquad \text{(no rule for 0)} \qquad \frac{\Delta \Vdash A^+}{\Delta \Vdash A^+ \oplus B^+} \qquad \frac{\Delta \Vdash B^+}{\Delta \Vdash A^+ \oplus B^+}$$

$$\text{(no rule for } \top\text{)} \qquad \frac{\Delta \Vdash \bullet A^-}{\Delta \Vdash \bullet A^- \& B^-} \qquad \frac{\Delta \Vdash \bullet B^-}{\Delta \Vdash \bullet A^- \& B^-} \qquad \frac{}{\cdot \Vdash \bullet \bot} \qquad \frac{\Delta_1 \Vdash \bullet A^- \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash \bullet A^- \parr B^-}$$

$$\frac{\Delta_1 \Vdash A^+ \quad \Delta_2 \Vdash \bullet B^-}{\Delta_1, \Delta_2 \Vdash \bullet A^+ \to B^-} \qquad \frac{}{A^- \Vdash {\downarrow} A^-} \qquad \frac{}{\bullet A^+ \Vdash \bullet {\uparrow} A^+}$$

Ψ_{pol} defines some standard polarized connectives, adopting the notation of linear logic. Most of the rules should be familiar from linear sequent calculus ($\bullet A$ can be read as $A^\perp$), but note that here we are only building *patterns,* and the linearity restriction will be dropped once we move to complete proofs and refutations. The connectives $\downarrow$ and $\uparrow$ deserve some explanation, being the only definitions here that insert anything directly into the frame. The "shifts" (introduced by Girard [28,29]) act as explicit coercions from negative to positive polarity ($\downarrow$) and vice versa ($\uparrow$). Having these coercions allows one to restrict other connectives to fixed polarities (e.g., $\otimes$ acts on a pair of positive propositions to produce a positive proposition) without any loss of expressivity.

Example 2. Consider a positive proposition P defined by a single paradoxical clause $\bullet P \Vdash P$, i.e., "P is true iff P is false". Then $P \prec \bullet P \prec P$, so $\prec_P$ is not well-founded.

Given a dictionary Ψ telling us how to interpret propositions by proof and refutation patterns, we can define full-blown proof and refutation in a generic way, essentially by the following informal recipes (applying the unbracketed instructions for positive polarity A, the bracketed instructions for negative polarity):

To prove [refute] A, find a proof [refutation] pattern for A and fill in its holes

To refute [prove] A, contradict every proof [refutation] pattern for A

Now let us make these recipes formal, assuming some fixed dictionary Ψ.

Definition 5 (Hypothetical judgments). *We write $\Gamma \vdash J$ to assert a judgment relative to a frame. In this case, we use the letter Γ to range over frames, calling Γ the* **context** *of J. Judgments J can be either A or $\bullet A$ (read as above), or Δ ("all of Δ") or $\#$ ("contradiction").*

We will explain the meaning of these hypothetical judgments through *canonical* derivations. That is, we define a very restricted system of inference rules, where each derivation can be seen as conveying a distinguished semantic content.

To prove a positive proposition, we choose one of its proof patterns, and derive the pattern's frame.

$$\frac{\Delta \Vdash A^+ \quad \Gamma \vdash \Delta}{\Gamma \vdash A^+}$$

Observe that because there can be many possible proof patterns for A^+ in Ψ, there can be many possible ways to apply this rule, and proving the proposition requires making a choice.

To refute a positive proposition, we must examine each of its proof patterns, and derive a contradiction from the pattern's frame.

$$\frac{\Delta \Vdash A^+ \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \bullet A^+}$$

Observe the similarity with Buchholz's Ω-rule. We apply here the same convention in interpreting the arrow above the line: the premise is satisfied just in case to every possible derivation of the left-hand side, we can give a derivation of the right-hand side. Note that there is only one possible way to apply this rule, given the dictionary definition of A^+. However, the rule can have many premises (indeed infinitely many), for however many A^+-patterns there are in the dictionary.

The rules of proof and refutation for negative propositions can be derived mechanically from the rules for positive polarity, by reversing the role of truth and falsehood.

To prove a negative proposition, we must examine each of its refutation patterns, and derive a contradiction from the pattern's frame. To refute a negative proposition, we choose one of its refutation patterns, and derive the pattern's frame.

$$\frac{\Delta \Vdash \bullet A^- \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A^-} \qquad \frac{\Delta \Vdash \bullet A^- \quad \Gamma \vdash \Delta}{\Gamma \vdash \bullet A^-}$$

The procedure for satisfying frames is straightforward: to satisfy a frame, we fill in all of its holes. The following rules simply unravel a frame down to its holes.

The empty frame is always satisfied; to satisfy the join of two frames, we must satisfy both frames.

$$\frac{}{\Gamma \vdash \cdot} \qquad \frac{\Gamma \vdash \Delta_1 \quad \Gamma \vdash \Delta_2}{\Gamma \vdash \Delta_1, \Delta_2}$$

Finally, we give two rules for deriving contradiction.

To derive contradiction from the context, we can find a positive proposition assumed to be false, and prove it, or we can find a negative proposition assumed to be true, and refute it.

$$\frac{\bullet A^+ \in \Gamma \quad \Gamma \vdash A^+}{\Gamma \vdash \#} \qquad \frac{A^- \in \Gamma \quad \Gamma \vdash \bullet A^-}{\Gamma \vdash \#}$$

Again, there can be many possible ways (though only finitely many) of applying these rules, one for every hole in the context.

Definition 6 (Canonical derivations). *A* **canonical** *derivation of $\Gamma \vdash J$ (relative to a dictionary Ψ) is a derivation of $\Gamma \vdash J$ applying only the above inference rules.*

There is a direct correspondence between canonical derivations relative to the dictionary Ψ_{pol} and standard presentations of focusing proofs for polarized classical sequent calculus with two-sided sequents [15, Ch. 3]. We are interested in arbitrary dictionaries, however, in particular ones containing propositions with infinitely many patterns, and possibly non-well-founded definitions. Therefore we make no assumption of finiteness on canonical derivations: they can be both infinitely branching and infinitely deep.

Given these core rules, we justify additional non-canonical rules by showing that they are admissible transformations on canonical derivations. For example, we can justify two important composition rules (corresponding to cuts in sequent calculus):

$$\frac{\Gamma \vdash A \quad \Gamma \vdash \bullet A}{\Gamma \vdash \#} \qquad \frac{\Gamma \vdash \Delta \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash \#}$$

It is easy to explain these rules as admissible transformations on canonical derivations, and to show that they produce well-founded derivations so long as $\prec_A$ and $\prec_\Delta$ are well-founded (see [15, §2.1.4]). In this way, we build up an operational semantics of proofs, which can also be seen as a "proof-theoretic semantics" [30] of programs. The semantics is in fact entirely deterministic, allowing for addition of side-effects and other computational phenomena, because canonical derivations intrinsically enforce continuation-passing style (cf. [15, Ch. 4]). We now illustrate this more concretely.

3 The Twelf encoding

We will follow the informal presentation of the preceding section closely, but with some variation, to build up the Twelf encoding.[4] First, because we want to take full advantage of higher-order abstract syntax, rather than explicitly mentioning contexts in the hypothetical judgments $\Gamma \vdash J$, we will instead leave Γ implicit as the LF context, with holes of Γ represented by LF variables. Moreover, rather than directly encoding canonical derivations, we will give rules for building non-canonical derivations, and define a (cut-elimination) procedure for reducing these derivations towards a normal form, driven by Twelf's logic programming engine.

We begin by restating Definitions 1–4 in terms that Twelf can understand:

```
pos : type. neg : type. tp : type.
⁺ : pos -> tp.  %postfix 10 ⁺.
⁻ : neg -> tp.  %postfix 10 ⁻.

frm : type.
· : frm.
, : frm -> frm -> frm.  %infix right 11 ,.
true : tp -> frm.       %postfix 9 true.
false : tp -> frm.      %postfix 9 false.

⊩ : frm -> frm -> type.  %infix right 8 ⊩.
```

Here we have declared `pos` and `neg` as LF (i.e., meta-level) types, standing for positive and negative (object-level) types.[5] We also declare an LF type `tp` to represent arbitrary polarized types, and `frm` to represent frames, with some simple constructors to build these up as inductive data structures. Finally, we declare the ambient dictionary as an LF type family $\Delta_1 \Vdash \Delta_2$. Note that we do *not* yet declare any specific inhabitants of `pos` or `neg`, let alone give their dictionary definitions. By default, Twelf treats LF types as open-ended, and allows new declarations to be interspersed throughout a file. Taking advantage of this feature and the modularity of the focusing analysis, we will first give the type-generic definition of the syntax and operational semantics, then introduce new types and patterns as they are needed to build examples.

```
j : type.
# : j.
⊢ : frm -> j -> j.   %infix right 8 ⊢.
sub : frm -> type. %abbrev val = [A] sub (A true). %abbrev con = [A] sub (A false).
exp : j -> type.
```

Here we declare the LF types which will be inhabited by *terms*, i.e., (non-canonical) derivations. As mentioned above, we want to use the implicit LF context to represent the context of hypothetical judgments, and so we write `val A`, `con A`, `sub` Δ, and `exp #`, instead of $\Gamma \vdash A$, $\Gamma \vdash \bullet A$, $\Gamma \vdash \Delta$, and $\Gamma \vdash \#$.[6] However, because

[4] We assume that the reader has some familiarity with LF and Twelf. Due to space limitations we omit some less interesting code below, but the full source is available from http://www.cs.cmu.edu/~noam/.

[5] I.e., object-level propositions. Because the Twelf encoding leads directly to an executable programming language, in this section we will freely employ the terminology of the "other side" of the propositions-as-types correspondence.

[6] Applying Curry-Howard, a derivation of Δ is called a *substitution*, and a derivation of $\#$ an *expression*. In the case of singleton substitutions, a proof of A is called a *value*, and a refutation of A is called a *continuation*.

we need the ability to bind multiple variables at a time (during pattern-matching), which is not directly representable by LF types, we define an auxiliary judgment $\Delta \vdash J$. The constructors for `exp (Δ ⊢ J)` simply introduce variables into the LF context one-by-one:[7]

```
λ· : exp J -> exp (· ⊢ J).
λ, : exp (XΔ1 ⊢ XΔ2 ⊢ J) -> exp (XΔ1 , XΔ2 ⊢ J).
λ : (sub XΔ -> exp J) -> exp (XΔ ⊢ J).
%prefix 9 λ·. %prefix 9 λ,. %prefix 9 λ.
```

Here the constructors $\lambda\cdot$ and `λ,` navigate the tree structure of the frame, while λ does the actual work of binding a variable. Now we give constructors for building terms in the implicit context. The constructor `v+` corresponds exactly to the canonical rule of positive proof, and `c-` to the canonical rule of negative refutation:

```
v+ : XΔ ⊩ A + true -> sub XΔ -> val (A +).
c- : XΔ ⊩ A - false -> sub XΔ -> con (A -).
```

The constructors `nil` and `++` correspond exactly to the canonical rules for deriving frames:

```
nil : sub ·.
++ : sub XΔ1 -> sub XΔ2 -> sub (XΔ1 , XΔ2).   %infix right 11 ++.
```

The constructors `throw` and `let` correspond to the composition principles at the end of Section 2:

```
throw : con A -> val A -> exp #.
let : sub XΔ -> exp (XΔ ⊢ #) -> exp #.
```

Let us throw in one additional non-logical expression:

```
abort : i -> exp #.
```

Here `i` is an LF type representing integers (definition omitted), and the expression `abort N` stands for a computation terminating with some return code. It is useful simply to give us some more interesting operational behavior to observe.

At last we come to the decisive moment: how to represent the Ω-rules? As we have suggested the trick boils down to defunctionalization. What that means here is that instead of giving a generic constructor for inhabiting the LF types `con (A +)` and `val (A -)` by functions from patterns to expressions, we will instead declare various constants of these types throughout the file, treating them as *tags,* and define an "apply function" that does the actual work of associating—for each tag—patterns with expressions. We call these functions `cbody` and `vbody`, since they define the body of the continuation/value denoted by each tag. Because the operational semantics of Twelf are based on logic programming, we declare them as relations:

```
cbody : con (A +) -> XΔ ⊩ A + true -> exp (XΔ ⊢ #) -> type.
vbody : val (A -) -> XΔ ⊩ A - false -> exp (XΔ ⊢ #) -> type.
%mode cbody +K +P -E.  %mode vbody +V +D -E.
```

Of course, since `cbody` and `vbody` are only declared as relations, they aren't forced to behave functionally—but with the appropriate keywords, Twelf will verify that they are in fact total functional relations, e.g.:[8]

```
%worlds () (cbody K P _). %total {K P} (cbody K P _). %unique cbody +K +P -E.
```

The check is actually trivial at this point, because we haven't declared any tags yet. Whenever we introduce a new tag (i.e., declare a typed value/continuation), we follow up by defining associated clauses of `[c/v]body`, and verify that `[c/v]body` remains `%total` and `%unique`. In this way, we induce Twelf to check that our pattern-matching definitions really are exhaustive and non-redundant (as we will illustrate with examples).

We will find it useful to consider the following pair of relations as well, duals to the above:

[7] Note that this kind of hybrid representation of contexts has been discussed by Crary [31].

[8] In functional languages built on LF, such as Delphin and Beluga, we would not have to go through these logic programming contortions, and could instead define `[c/v]body` as real (computational) functions. Incidentally, defunctionalization in the context of logic programming was also considered by Warren [32].

```
vfact : val (A +) -> XΔ ⊩ A + true -> sub XΔ -> type.
cfact : con (A -) -> XΔ ⊩ A - false -> sub XΔ -> type.
%mode vfact +V -P -Xσ.  %mode cfact +K -D -Xσ.
```

These can be seen as lemmas, stating the factorization property of positive values and negative continuations (i.e., that they split into a pattern and a substitution). Again, at this point they hold trivially, because we only gave the canonical rules `v+` and `c-`. However, these relations give us a means of introducing additional, non-canonical constructors for `val (A +)` and `con (A -)`, so long as we can justify them by maintaining the `vfact` and `cfact` properties.

Given this generic definition of syntax, we can define a generic big-step operational semantics:

```
result : type.
halt : i -> result.
eval : exp # -> result -> type.  %mode eval +E -R.
```

To define (the Twelf logic program) `eval`, we first need (another logic program) `load`:

```
load : sub XΔ -> exp (XΔ ⊢ J) -> exp J -> type.  %mode load +Xσ +T -T'.
ld/nil : load nil (λ· T) T.
ld/++ : load (Xσ_1 ++ Xσ_2) (λ, T) T'' <- load Xσ_1 T T' <- load Xσ_2 T' T''.
ld/x : load Xσ (λ T*) (T* Xσ).
```

Then we can define `eval` in four easy clauses:

```
ev/throw+ : eval (throw K V) R <- vfact V P Xσ <- cbody K P E <- eval (let Xσ E) R.
ev/throw- : eval (throw K V) R <- cfact K D Xσ <- vbody V D E <- eval (let Xσ E) R.
ev/let :  eval (let Xσ E) R <- load Xσ E E' <- eval E' R.
ev/abort : eval (abort N) (halt N).
```

The really interesting clauses here are `ld/x` in the definition of `load`, and `ev/throw`$^+$ and `ev/throw`$^-$ in the definition of `eval`, which all take advantage of the "higher-orderness" of the syntax, in the different senses of "higher-order". The `load` clause uses HOAS in the traditional sense, applying LF-level substitution to perform substitution on terms. The two `eval` clauses, on the other hand, appeal to Twelf-level logic programming to perform the application of defunctionalized Ω-rules. We elide here the keywords instructing Twelf to verify that the definition of `eval` is exhaustive and deterministic.

With the generic portion of the language definition complete, we now move to particular examples. We begin by defining some of the standard polarized connectives (cf. Example 1):

```
1 : pos.
  #* : · ⊩ 1 + true.
⊗ : pos -> pos -> pos.  %infix right 14 ⊗.
  #, : XΔ_1 ⊩ A + true -> XΔ_2 ⊩ B + true -> (XΔ_1 , XΔ_2) ⊩ A ⊗ B + true.  %infix right 11 #,.
0 : pos.
⊕ : pos -> pos -> pos.  %infix right 13 ⊕.
  #Inl : XΔ ⊩ A + true -> XΔ ⊩ A ⊕ B + true.
  #Inr : XΔ ⊩ B + true -> XΔ ⊩ A ⊕ B + true.
→ : pos -> neg -> neg.  %infix right 12 →.
  #; : XΔ_1 ⊩ A + true -> XΔ_2 ⊩ B - false -> (XΔ_1 , XΔ_2) ⊩ A → B - false.  %infix right 11 #;.
& : neg -> neg -> neg.  %infix right 13 &.
  #Fst : XΔ ⊩ A - false -> XΔ ⊩ A & B - false.
  #Snd : XΔ ⊩ B - false -> XΔ ⊩ A & B - false.
↓ : neg -> pos.                  ↑ : pos -> neg.
  _v : A - true ⊩ ↓ A + true.      _k : A + false ⊩ ↑ A - false.
```

Observe that whenever we declare a new (object-level) type constructor, we also introduce a set of pattern constructors which define the meaning of the type. Thus the dictionary entries for 1, $\otimes$, 0, and $\oplus$ look very much like standard datatype definitions. (The definitions of `&` and $\to$ appear less conventional because they are stated in terms of continuation patterns, but this is simply a dual principle.) We can likewise define constructors for general recursive types (note the use of higher-order abstract syntax):

```
rec+ : (pos -> pos) -> pos.
  #Fold : XΔ ⊩ A (rec+ A) + true -> XΔ ⊩ rec+ A + true.
rec- : (neg -> neg) -> neg.
  #Unfold : XΔ ⊩ A (rec- A) - false -> XΔ ⊩ rec- A - false.
```

Using these type constructors, we can derive some standard inductive and coinductive types, such as `nat = rec+ [X] 1 ⊕ X` and `stream A = rec- [X] A & X`. Finally, along different lines, we define `int`:

```
int : pos.
  #I : i -> · ⊩ int + true.
```

Recall that `i` is the LF-level type of integers. The object-level type `int` therefore supports "native arithmetic", so to speak, with different relations (i.e., Twelf logic programs) on `i` (such as `add : i -> i -> i -> type` and `mult : i -> i -> i -> type`, etc.) serving as a "foreign-function interface" for `int`. We will illustrate this shortly, but to warm up let us first define a very simple but useful continuation:

```
exit : con (int +).
exit/_ : cbody exit (#I N) (λ· abort N).
```

`exit` is the `int`-continuation that reads a native integer and aborts with that integer. Again observe the overall form of the definition: first we declare a constant `K : con (A +)`, and then we give clauses inhabiting `cbody K P1 E1`, `cbody K P2 E2`, etc. (Here, we only used a single `cbody` clause, which we can verify is enough with the appropriate `%total` and `%unique` checks.)

Now we define `plus`, a curried function (negative value) on `int`s in continuation-passing style:

```
plus : val (int → int → ↑ int -).
plus/_ : vbody plus ((#I M) #; (#I N) #; _k) (λ, λ· λ, λ· λ [k] throw k (v+ (#I P) nil))
             <- add M N P.
```

Modulo the bureaucratic string of λ's, this definition is dead simple, just appealing to the meta-operation `add` (with the standard logic programming definition). (Compare it with the definition further below of `plus*`, the corresponding curried function on the recursive datatype `nat`.)

Before we move on, we can try evaluating a program (here `z : i` and `s : i -> i` and `i2 = s s z`):

```
%query 1 * eval (throw (c- ((#I i2) #; (#I i2) #; _k) (nil ++ nil ++ exit)) plus) R.
```

Indeed the Twelf server answers `R = halt (s s s s z)`, although it might be hard recognizing the question in this syntax ("$2 + 2 = ?$"). For more perspicuous programs, we can start building up some syntactic sugar. Because working directly with values in factorized form can get cumbersome, we declare additional value constructors for the positive connectives, such as:

```
, : val (A +) -> val (B +) -> val (A ⊗ B +). %infix none 9 ,.
inl : val (A +) -> val (A ⊕ B +).  inr : val (B +) -> val (A ⊕ B +).
fold : val (A (rec+ A) +) -> val (rec+ A +).
```

But it is not enough to stipulate these constructors—we must *justify* them by extending `vfact`, e.g., by:

```
,/_ : vfact (V1 , V2) (P1 #, P2) (Xσ1 ++ Xσ2) <- vfact V1 P1 Xσ1 <- vfact V2 P2 Xσ2.
```

Using these new constructors we can also derive some additional syntactic sugar, e.g.:

```
zz : val (nat +) = fold (inl (v+ #* nil)).
ss : val (nat +) -> val (nat +) = [n] fold (inr n).  %prefix 9 ss.
```

Next we introduce some syntactic sugar for building negative values and positive continuations, in defunctionalized style. Here we define the continuation constructor μ~:[9]

[9] With the name inspired by [33]—the reader can try defining the dual constructor μ (or consult the online source).

```
μ~ : (val (A +) -> exp #) -> con (A +).
μ~/_ : cbody (μ~ K) P (λ [σ] K (v+ P σ)).
```

Note that because μ~ is a higher-order LF constructor, we will use it precisely when we *don't* need to do any pattern-matching on the argument, i.e., when the body of the continuation can be defined parametrically. The following constructors provide sugar for function abstraction and application:[10]

```
fn : (val (A +) -> val (B -)) -> val (A → B -).
fn/_ : vbody (fn V) (P #; D) (λ, λ [σ1] λ [σ2] let σ2 (E (v+ P σ1)))
        <- {x} vbody (V x) D (E x).
@ : val (A → B -) -> val (A +) -> val (B -).  %infix left 11 @.
@/_ : vbody (F @ V) D (λ [σ] let (Xσ0 ++ σ) E) <- vfact V P Xσ0 <- vbody F (P #; D) E.
```

And finally, a useful bit of syntactic sugar for plugging a value of type $\uparrow A$ with a continuation:

```
$ : val (↑ A -) -> con (A +) -> exp # = [v] [k] throw (c- _k k) v.   %infix none 10 $.
```

Using some of this sugar, we define `plus*` (here `#Zz` and `#Ss` are the derived pattern constructors for `nat`):

```
plus* : val (nat → nat → ↑ nat -).
plus*/z : vbody plus* (#Zz #; N #; _k) (λ, λ· λ, λ [σ] λ [k]
                throw k (v+ N σ)).
plus*/s : vbody plus* ((#Ss M) #; N #; _k) (λ, λ [σ1] λ, λ [σ2] λ [k]
                plus* @ (v+ M σ1) @ (v+ N σ2) $ μ~ [x] throw k (ss x)).
```

Except for some uninteresting contortions to bind substitution variables and carry them through, the body of `plus*` corresponds to the standard recursive definition of addition in continuation-passing style. Observe that for the first time we are using two separate clauses to define a defunctionalized value, applying case-analysis on the pattern. It goes without saying that Twelf verifies this definition is `%total` and `%unique` (but would complain, say, if we left out the `plus*/z` clause).

This time we try asking "2 + 3 = ?", hopefully in more recognizable form (to convert the result of `plus*` into a native `int`, we use a continuation transformer `nat2int : con (int +) -> con (nat +)`, whose definition is omitted here):

```
%query 1 * eval (plus* @ (ss ss zz) @ (ss ss ss zz) $ nat2int exit) R.
```

And indeed, Twelf answers that `R = halt (s s s s s z)`.

To end this section, we will work up to a more significant example, building macros that let us view direct-style programs in call-by-value lambda calculus (with products, sums, natural numbers, an effectful abort command, and left-to-right evaluation order) as syntactic sugar for focusing proofs—or, if you prefer, we build a continuations semantics of lambda calculus via pattern-based focusing. Lambda calculus values will be represented simply by values of positive type, while arbitrary terms will be represented by negative values of ↑'d type:

```
%abbrev vl : pos -> type = [A] val (A +).
%abbrev tm : pos -> type = [A] val (↑ A -).
```

As mentioned in the footnote, we decompose the call-by-value function space like so:

```
→v : pos -> pos -> pos = [A] [B] ↓ (A → ↑ B).  %infix right 12 →v.
```

Now we give some standard elimination forms for products and sums, as continuation constructors:

```
split : (val (A +) -> val (B +) -> exp #) -> con (A ⊗ B +).
split/_ : cbody (split E) (P1 #, P2) (λ, λ [σ1] λ [σ2] E (v+ P1 σ1) (v+ P2 σ2)).
case : con (A +) -> con (B +) -> con (A ⊕ B +).
```

[10] Note that → is neither call-by-value nor call-by-name, but a more primitive function space. Below we explain the call-by-value decomposition, $A \xrightarrow{v} B = {\downarrow}(A \to {\uparrow}B)$ [15, Ch. 4].

```
case/inl : cbody (case K1 K2) (#Inl P) E1 <- cbody K1 P E1.
case/inr : cbody (case K1 K2) (#Inr P) E2 <- cbody K2 P E2.
```

as well as a control operator `letcc` for introducing ↑'d values, and a continuation constructor `force` for eliminating ↓'d values:

```
letcc : (con (A +) -> exp #) -> val (↑ A -).
letcc/_ : vbody (letcc E) _k (λ [k] E k).
force : (val (A -) -> exp #) -> con (↓ A +).
force/_ : cbody (force E) _v (λ [x] E x).
```

Finally, we define the embedding of direct-style as a collection of macros for building `tm`s:

```
! : vl A -> tm A = [x] letcc [k] throw k x.  %prefix 9 !.
Pair : tm A -> tm B -> tm (A ⊗ B)
     = [e1] [e2] letcc [k] e1 $ μ~ [x1] e2 $ μ~ [x2] throw k (x1 , x2).
Fst : tm (A ⊗ B) -> tm A = [e] letcc [k] e $ split [x] [y] throw k x.
Snd : tm (A ⊗ B) -> tm B = [e] letcc [k] e $ split [x] [y] throw k y.
Inl : tm A -> tm (A ⊕ B) = [e] letcc [k] e $ μ~ [x] throw k (inl x).
Inr : tm B -> tm (A ⊕ B) = [e] letcc [k] e $ μ~ [x] throw k (inr x).
Case : tm (A ⊕ B) -> (vl A -> tm C) -> (vl B -> tm C) -> tm C
     = [e] [f] [g] letcc [k] e $ case (μ~ [x] f x $ k) (μ~ [y] g y $ k).
Fn : (vl A -> tm B) -> tm (A →v B) = [f] ! v+ _v (fn [x] letcc [k] f x $ k).
App : tm (A →v B) -> tm A -> tm B = [f] [e] letcc [k] f $ force [f0] e $ μ~ [x] f0 @ x $ k.
Z : tm nat = ! zz.
S : tm nat -> tm nat = [e] letcc [k] e $ μ~ [x] throw k (ss x).  %prefix 9 S.
Plus : tm nat -> tm nat -> tm nat
     = [e1] [e2] letcc [k] e1 $ μ~ [n1] e2 $ μ~ [n2] plus* @ n1 @ n2 $ k.
Abort : tm nat -> tm A = [e] letcc [k] e $ nat2int exit.
```

For convenience, we define an LF type abbreviation `run E N = eval (E $ nat2int exit) (halt N)`, representing the fact that a `nat`-computation terminates with result `n`. We can now run direct-style programs directly. For example, Twelf answers all of the following queries correctly:

```
%query 1 * run (Plus (S S Z) (S S S Z)) N1.              % N1 = s s s s s z
%query 1 * run (Plus (S S Z) (Abort (S Z))) N2.          % N2 = s z
%query 1 * run (Plus (Abort Z) (Abort (S Z))) N3.        % N3 = z
%query 1 * run (App (Fn [x] Plus (! x) (S Z)) (S Z)) N4. % N4 = s s z
```

And this is all still accomplished in an entirely generic way! Since the four clauses given a few pages back, we have not touched the definition of `eval`.

4 Conclusions

I hope to have offered some intriguing examples of the power of pattern-based focusing, particularly when combined with traditional higher-order abstract syntax, and to have shown that Ω-rules need not be so mystical when viewed through the lens of defunctionalization. Obviously, I would like to get a better theoretical understanding of the precise capabilities of this representation technique. Considerable work already exists in the field of infinitary proof theory on reading finite derivations as notation systems for infinite derivations—particularly by Mints [34,35], Buchholz [36], and Schwichtenberg [37]—and it could be profitable to translate their results to the realm of focusing proofs and the Curry-Howard correspondence.

Acknowledgements. Many thanks to the anonymous PSTT reviewer for detailed and insightful suggestions which led to the improvement of both the informal presentation and the Twelf code.

References

1. Gentzen, G.: Die Widerspruchfreiheit der reinen Zahlentheorie. Mathematische Annalen **112** (1936) 495–565 English translation in M. E. Szabo, editor, *The Collected Papers of Gerhard Gentzen*, North-Holland, 1969.

2. Schütte, K.: Beweistheoretische Erfassung der unendliche Induction in der Zahlentheorie. Mathematische Annalen **122** (1950) 369–389
3. Hilbert, D.: Die Grundlegung der elementaren Zahlenlehre. Mathematische Annalen **104** (1931) 485–494
4. Novikov, P.: On the consistency of a certain logical calculus. Matématicésky sbovnik **12**(3) (1943) 353–369
5. Lorenzen, P.: Algebraische und logistische untersuchungen über freie Verbände. Journal of Symbolic Logic **16** (1951) 81–106
6. Shoenfield, J.R.: On a restricted ω-rule. Bulletin de l'academie Polonaise des sciences **VII**(7) (1959)
7. Buchholz, W., Feferman, S., Pohlers, W., Sieg, W.: Iterated Inductive Definitions and Subsystems of Analysis: Recent Proof-Theoretical Studies. Springer-Verlag (1981)
8. Prawitz, D.: Natural Deduction: A Proof Theoretical Study. Almquist and Wiksell, Stockholm (1965)
9. Girard, J.Y.: On the unity of logic. Annals of Pure and Applied Logic **59**(3) (1993) 201–217
10. Zeilberger, N.: On the unity of duality. Annals of Pure and Applied Logic **153**(1) (2008) Special issue on "Classical Logic and Computation".
11. Zeilberger, N.: Focusing and higher-order abstract syntax. In: ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. (January 2008) 359–369
12. Licata, D.R., Zeilberger, N., Harper, R.: Focusing on binding and computation. In: IEEE Symposium on Logic in Computer Science. (2008)
13. Licata, D.R., Harper, R.: Positively dependent types. In: ACM SIGPLAN-SIGACT Workshop on Programming Languages Meets Program Verification. (January 2009)
14. Zeilberger, N.: Refinement types and computational duality. In: ACM SIGPLAN-SIGACT Workshop on Programming Languages Meets Program Verification. (January 2009)
15. Zeilberger, N.: The logical basis of evaluation order and pattern-matching. PhD thesis, Carnegie Mellon University (April 2009) Computer Science Department.
16. Andreoli, J.M.: Logic programming with focusing proofs in linear logic. Journal of Logic and Computation **2**(3) (1992) 297–347
17. Escardó, M.H.: Infinite sets that admit fast exhaustive search. In: IEEE Symposium on Logic in Computer Science, IEEE Computer Society (2007) 443–452
18. Reynolds, J.C.: Definitional interpreters for higher-order programming languages. In: ACM '72: Proceedings of the ACM annual conference. (1972) 717–740
19. Danvy, O., Nielsen, L.R.: Defunctionalization at work. In: ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, ACM (2001) 162–174
20. Danvy, O., Millikin, K.: Refunctionalization at work. BRICS Report RS-07-7 (revised version), University of Aarhus (March 8 2007)
21. Nordström, B., Petersson, K., Smith, J.M.: Programming in Martin-Löf's Type Theory. Volume 7 of International Series of Monographs on Computer Science. Clarendon Press, Oxford, England (1990)
22. Pfenning, F., Schürmann, C.: System Description: Twelf — a meta-logical framework for deductive systems. In Ganzinger, H., ed.: Proceedings of the 16th International Conference on Automated Deduction (CADE-16), Trento, Italy, Springer-Verlag LNAI 1632 (July 1999) 202–206
23. Harper, R., Honsell, F., Plotkin, G.: A framework for defining logics. Journal of the ACM **40**(1) (1993) 143–184
24. Pfenning, F., Elliott, C.: Higher-order abstract syntax. In: Programming Language Design and Implementation. (1988) 199–208
25. Pientka, B., Dunfield, J.: Programming with proofs and explicit contexts. In Antoy, S., Albert, E., eds.: ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, ACM (2008) 163–173
26. Poswolsky, A., Schürmann, C.: Practical programming with higher-order encodings and dependent types. In: European Symposium on Programming. (2008)
27. Takeuti, G.: Proof Theory. Volume 81 of Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam (1975)
28. Girard, J.Y.: On the sex of angels (1991) Post to LINEAR mailing list, December 6, 1991, archived at http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html.
29. Girard, J.Y.: Locus solum: From the rules of logic to the logic of rules. Mathematical Structures in Computer Science **11**(3) (2001) 301–506
30. Kahle, R., Schroeder-Heister, P., eds.: Proof-Theoretic Semantics. Special issue of *Synthese* (2006)
31. Crary, K.: Explicit contexts in LF (extended abstract). Electr. Notes Theor. Comput. Sci **228** (2009) 53–68
32. Warren, D.H.D.: Higher-order extensions to Prolog: Are they needed? In: Machine Intelligence 10. Halsted Press (1982) 441–454
33. Curien, P.L., Herbelin, H.: The duality of computation. In: ACM SIGPLAN International Conference on Functional Programming. Volume 35(9) of SIGPLAN Notices. ACM Press (2000) 233–243
34. Mints, G.E.: Finite investigations of transfinite derivations. Journal of Soviet Mathematics **10** (1978) 548–596 Appears in Grigori Mints, *Selected papers in Proof Theory.* Bibliopolis, 1992.
35. Mints, G.E.: Reduction of finite and infinite derivations. Annals of Pure and Applied Logic **104**(1–3) (2000) 167–188
36. Buchholz, W.: Notation systems for infinitary derivations. Archive for Mathematical Logic **30** (1991) 277–296
37. Schwichtenberg, H.: Finite notations for infinite terms. Annals of Pure and Applied Logic **94**(1-3) (1998) 201–222