Dependently Typed Programming with Domain-Specific Logics (Thesis Proposal DRAFT)

Daniel R. Licata
Carnegie Mellon University
drl@cs.cmu.edu

October 6, 2008

Abstract

We propose a thesis defending the following statement:

The logical notions of polarity and focusing provide a foundation for dependently typed programming with domain-specific logics, with applications to certified software and mechanized metatheory.

1 Introduction

Type systems have proved to be effective and scalable formal methods. Explicating the type structure of a program has both short- and long-term benefits: In the short-term, types help programmers write working code quickly; as ML and Haskell programmers know, it is often the case that once you set up the types properly, the code writes itself. Of course, most of software development happens well after the first programmer writes the first version of a piece of code: more time is spent evolving old code to meet new circumstances, and combining programs into larger systems. Types help with the long-term tasks of software reuse and evolution because they give a foundation for modularity: a programmer can specify the interface of a component without revealing its implementation. Interfaces limit the coupling of a a system's components and provide formal guidance about how components can be composed and evolved. They also serve as machine-checked documentation, helping new programmers understand the pieces of a software system: types explain how and why a program works.

However, some of the effectiveness and scalability of type systems has been the result of concentrating on relatively simple properties that admit automated verification with little programmer input. One approach to lifting these limitations has been to design domain-specific type systems such as the following:

- Cryptol [29] is a language for implementing cryptographic protocols. These algorithms typically involve complex operations on bit vectors of various lengths—

e.g., concatenating four 8-bit words to make a 32-bit word. Cryptol's type system tracks the lengths of vectors statically, providing simple constraints on these operations that rule out many programming errors.

- Many programs must manipulate sensitive resources (e.g., a user's information in a database) in a secure manner. Security-typed programming languages such as Aura [38] and PCML5 [6] employ authorization logics to statically prevent unauthorized access to controlled resources.

- Ynot [50], an implementation of Hoare Type Theory [48], provides a separation logic for reasoning about imperative code.

These language's type systems are *domain-specific logics* (DSLs) for reasoning about a particular programming style or application domain. Domain-specific logics extend the class of properties that programmers can specify and verify. Moreover, they offer a range of verification options, from simple logics for simple safety checks (as in Cryptol), which often admit effective decision procedures, to sophisticated logics for proving full correctness (as in Ynot), which require non-trivial proofs.

Despite these advantages, adopting new application-specific languages has costs, such as the engineering effort required to build and maintain the infrastructure supporting a new language—documentation, libraries, a fast compiler—and the time spent training new programmers to use the language. Additionally, treating each domain-specific logic in isolation makes it difficult for programs written using different domain-specific type systems to be composed—e.g., if an application both implements a cryptographic protocol and manipulates sensitive resources. These costs can be mitigated by giving a single *host language* inside of which various domain-specific logics can be constructed: one engineering effort is shared by all DSLs; programmers need only learn new libraries; and modules using different DSLs can be composed.

The central contribution of this thesis will be a new host language for programming with domain-specific logics. This language will make it easy for programmers to define logical systems, reason about them, and use them to reason about code. It will have applications both to *certified software*—using DSLs to reason about code—and *mechanized metatheory*—formalized reasoning about DSLs, combining the advantages of functional programming languages such as ML and Haskell with proof assistants such as Twelf [53] and Coq [18].

To program with domain specific logics, we require a language rich enough to represent and compute with logical systems. Our approach to representing logical systems builds on LF [34], a logical framework providing two main ingredients: dependent types, which are necessary to adequately represent the deductive apparatus of logical systems, and a function space for representing binding and scope—bound variables, α-conversion, and substitution at the level of syntax, and hypothetical judgements, such as the consequence relation of a logic, at the level of proofs. While LF functions are suitable for representing variable binding, they provide no account of computation with logical systems, as is provided by the function space of Agda and Coq or ML and Haskell. Consequently, it is necessary to combine LF with some further mechanism for computation, such as the separate computational languages of Twelf [53], Delphin [56], and Beluga [54].

In previous work [42], we began to investigate an alternative approach, using the logical notions of polarity [32] and focusing [4] to integrate representational and computational functions as two types in a single, simply-typed, logical framework. Representational functions adequately represent binding, whereas computational functions allow computation by structural recursion on syntax and derivations. This integrated approach provides a novel and rich logical framework, allowing inference rules that mix iterated inductive definitions [44] (using computational functions) and hypothetical judgements (using representational functions).

Technically, our work exploits the Curry-Howard correspondence between focused proofs and pattern-matching functional programs, following Zeilberger's higher-order formulation of focusing [72, 73]. This formalism has three key features: First, the syntax of programs reflects the interplay of *focus* (choosing patterns) and *inversion* (pattern matching), with individual types defined by their pattern typing rules. Second, the syntax of types is *polarized*, distinguishing positive data (introduced by focus, eliminated by inversion) from negative computation (introduced by inversion, eliminated by focus). This provides a natural framework for integrating representations of logics (as positive types) and computation with them (as negative types). Third, pattern matching is represented abstractly by *meta-functions*— functions in the ambient mathematical system in which our type theory itself is defined—from patterns to expressions (hence *higher-order focusing*), and the syntax and typing rules of our type theory are defined by iterated inductive definitions [44]. This allows our type theory to be computationally open-ended (cf. Howe [37]) with respect to the meaning of pattern-matching—any method of transforming every pattern for A into an expression of type B counts as a pattern-match from A to B—and affords the freedom to use several different notations for pattern matching in a single program, and to import functions from other languages and systems.

While this previous work has shown promise, it requires a number of extensions to make a practical language for programming with domain-specific logics.

Proposed Work

This thesis will support the following statement:

> **Thesis Statement:** *The logical notions of polarity and focusing provide a foundation for dependently typed programming with domain-specific logics, with applications to certified software and mechanized metatheory.*

This statement has both a theoretical component and a practical component. To justify that the logical notions of polarity and focusing provide a foundation for dependently typed programming with domain-specific logics, we will define a type theory with full support for programming with domain-specific logics, presented in the polarized style described above. Relative to our previous work, this type theory will add the following features:

- **Dependent types.** Our previous work considers only a simply-typed framework, which is not expressive enough for representing deductive systems. A major component of the proposed work is adding dependent types to this calculus.

- **Effects and Modules.** Our previous work is a simple core calculus, which lacks support for many common programming features, such as computational effects and modules. We will investigate whether the polarized formalism offers any new insights into the type theoretic accounts of these features.

To justify that this language has applications to certified software and mechanized metatheory, we will build a prototype implementation and program several examples. To do so, we must bridge the gap between the above type theory and a usable implementation:

- **Syntax for meta-functions.** Higher-order focusing leaves pattern matching abstract, relying on an infinitary representation using meta-level functions. In practice, we must give a traditional, finitary account of these meta-functions as a basis for an implementation.

- **Type and term reconstruction.** To make programming practical, we must consider conveniences such as type and term reconstruction.

In the remainder of this thesis proposal, we discuss the theoretical (Section 2) and practical (Section 3) components of the proposed work in more detail.

2 Theory

2.1 Dependent types

Our previous work [42] considers only a simply-typed framework, which is not expressive enough for representing deductive systems. This limitation can be addressed by adding dependent types—types that contain programs. However, adding dependency to a programming language is difficult, chiefly because type checking a dependently typed language requires comparing programs for equality. This places constraints on the run-time programming language: for example, it is much harder to reason about the equality of programs that utilize storage effects than of purely functional programs. To manage these difficulties, it is useful to consider restricted forms of dependency, rather than choosing *a priori* to allow dependency on all run-time programs. We can consider three levels of dependency:

1. **Dependency on framework data** First, we may scale back from the integrated approach to binding and computation described above, and take LF "off the shelf" as a data-level representational framework, with an entirely separate computational language. This approach, which is taken by Twelf, Delphin, and Beluga, is simple, as it requires comparing only LF terms for equality. However, it sacrifices the advantages of mixing binding and computation that we have discussed in previous work. As a baseline, we show below how to account for this form of dependency in our polarized calculus.

2. **Dependency on purely positive data** In the thesis, we will consider a simple generalization, based on the observation that it should not be too complicated to

allow dependency on positive *data*; it is only allowing dependency on negative *computations* that creates complications. This approach will allow dependency on any purely positive data (types with no negative subcomponents). It permits inference rules that mix binding and computation, while restricting the subjects of those rules in order to avoid the complications arising from full dependency on computation.

3. **Full dependency** More ambitiously, we may consider a type theory that permits full dependency on both data and computation. The primary advantage of full dependency is that it allows after-the-fact verification of computations, using dependency to state properties of them.

To establish a baseline for the thesis work, and to introduce our polarized formalism, we now consider the first form of dependency in detail. This type theory consists of:

- A representational language, the LF logical framework.
- A computational language based on polarized intuitionistic logic. The computational language is specified by:
  - Defining its types (Figure 1) and patterns (Figure 3).
  - A focusing framework (Figure 4) and its operational semantics (Figure 5)

We discuss LF in Section 2.1.1, types and patterns in Section 2.1.2, and the focusing framework in Section 2.1.5.

2.1.1 LF

We briefly review the LF methodology for representing languages and logics [34]: LF generalizes the ML datatype mechanism with (1) dependent types and (2) support for binding and scope. The judgements of a domain-specific logic (DSL) are represented as LF types, where dependency is used to ensure adequacy. Derivations in a DSL are represented as canonical (β-normal, η-long) LF terms. LF function types are used to represent binding and scope, including the bound variables of DSL syntax and the contexts of DSL hypothetical judgements. Structural induction over canonical LF terms corresponds to induction over DSL syntax and derivations: inductive proofs about a DSL can be recast as proofs by induction on the the LF representation.

We use a presentation of LF with with syntax for canonical forms only [68]:

LF kind	K	::=	$\mathsf{type} \mid \Pi\, u{:}A.\, K$
LF type	A	::=	$a\ M_1 \ldots M_n \mid \Pi\, u{:}A_1.\, A_2$
LF term	M	::=	$u\ M_1 \ldots M_n \mid \lambda\, u.\, M$
LF signature	Σ	::=	$\cdot \mid \Sigma, a : K \mid \Sigma, u : A$
LF context	Ψ	::=	$\cdot \mid \Psi, u : A$
LF world	$\mathcal{W}$	::=	$\{\Psi_1, \ldots\}$

All LF judgements are tacitly parametrized by a fixed signature Σ. In the following, we will make use of the judgements:

- $\Psi \vdash_{\text{LF}} A\ \mathsf{type}$ The type A is a well-formed in Ψ
- $\Psi \vdash_{\text{LF}} M : A$ The term M is a canonical form of type A in Ψ
- $\Psi \vdash \Psi' \in \mathcal{W}$ The context Ψ' is in the world (set of contexts) $\mathcal{W}$. This judgement also ensures that $\vdash_{\text{LF}} \Psi, \Psi'\ \mathsf{ctx}$, i.e., that the context Ψ, Ψ' is well-formed.

We refer the reader to the literature for the definitions of these judgements: Watkins et al. [68] discuss type formation and typing; one possible definition of worlds $\mathcal{W}$ is the regular worlds notation of Twelf [53].

2.1.2 Polarity and Focusing

Natural deduction is organized around introduction and elimination: For example, the disjoint sum type $A \oplus B$ is introduced by constructors inl and inr and eliminated by pattern-matching; the computational function type $A \to B$ is introduced by pattern-matching on the argument A and eliminated by application. Polarized logic [4, 31, 39, 41, 72] partitions types into two classes, called *positive* (notated A^+) and *negative* (notated A^-). Positive types, such as $\oplus$, are introduced by choice and eliminated by pattern-matching, whereas negative types, such as $\to$, are introduced by pattern-matching and eliminated by choice. More specifically, positive types are *constructor-oriented*: they are introduced by choosing a constructor, and eliminated by pattern matching against constructors, like datatypes in ML. Negative types are *destructor-oriented*: they are eliminated by choosing an an observation, and introduced by pattern-matching against all possible observations ($A \to B$ is observed by supplying a value of type A, and therefore defined by matching against such values). Choice corresponds to Andreoli's notion of *focus*, and pattern-matching corresponds to *inversion*. These distinctions can be summarized as follows:

	introduce A	eliminate A
A is positive	by focus	by inversion
A is negative	by inversion	by focus

In higher-order focusing [42, 72, 73], types are specified by patterns, which are used in both focus and inversion: focus phases choose a pattern, whereas inversion phases pattern-match. In this section, we define the types and patterns of our language—constructor patterns for positive types, and destructor patterns for negative types. Note that patterns must be defined prior to the focusing framework presented in Section 2.1.5, which uses an iterated inductive definition quantifying over them to specify inversion.

2.1.3 Types

We present the rules for type formation in Figure 1. The judgements $\langle \Psi \rangle\ A^+\ \mathsf{type}$ and $\langle \Psi \rangle\ A^-\ \mathsf{type}$ define the well-formed types, which are considered relative to an LF context Ψ. The basic positive types of polarized type theory are products ($A^+ \otimes B^+$ and 1), sums ($A^+ \oplus B^+$ and 0), and shift ($\downarrow A^-$), the inclusion of negative types into positive types. The formation rules for these types carry the LF context Ψ through unchanged.

$$
\begin{array}{llll}
\text{Pos. type} & A^+ & ::= & \downarrow A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+ \\
& & & \mid \exists_A(\tau^+) \mid \Psi \Rightarrow A^+ \mid \Box A^+ \mid \exists_{\mathcal{W}}(\psi^+) \\
& & & \text{where } \tau^+ ::= \{\, M \mapsto A^+ \mid \ldots \} \\
& & & \qquad \psi^+ ::= \{\, \Psi \mapsto A^+ \mid \ldots \} \\
\text{Neg. type} & A^- & ::= & \uparrow A^+ \mid A^+ \to B^- \mid \top \mid A^- \& B^- \\
& & & \mid \forall_A(\tau^-) \mid \Psi \curlywedge A^- \mid \diamond A^- \mid \forall_{\mathcal{W}}(\psi^-) \\
& & & \text{where } \tau^- ::= \{\, M \mapsto A^- \mid \ldots \} \\
& & & \qquad \psi^- ::= \{\, \Psi \mapsto A^- \mid \ldots \} \\
\text{CPT} & C^+ & ::= & \langle \Psi \rangle\, A^+ \\
\text{CNT} & C^- & ::= & \langle \Psi \rangle\, A^-
\end{array}
$$

$\boxed{\langle \Psi \rangle\, A^+\ \mathsf{type}}$

$$\frac{\langle \Psi \rangle\, A^-\ \mathsf{type}}{\langle \Psi \rangle \downarrow A^-\ \mathsf{type}} \qquad \frac{}{\langle \Psi \rangle\, 1\ \mathsf{type}} \qquad \frac{\langle \Psi \rangle\, A^+\ \mathsf{type} \quad \langle \Psi \rangle\, B^+\ \mathsf{type}}{\langle \Psi \rangle\, A^+ \otimes B^+\ \mathsf{type}}$$

$$\frac{}{\langle \Psi \rangle\, 0\ \mathsf{type}} \qquad \frac{\langle \Psi \rangle\, A^+\ \mathsf{type} \quad \langle \Psi \rangle\, B^+\ \mathsf{type}}{\langle \Psi \rangle\, A^+ \oplus B^+\ \mathsf{type}}$$

$$\frac{\Psi \vdash_{\text{LF}} A\ \mathsf{type} \quad (\Psi \vdash_{\text{LF}} M : A \ \longrightarrow\ \langle \Psi \rangle\, \tau^+(M)\ \mathsf{type})}{\langle \Psi \rangle\, \exists_A(\tau^+)\ \mathsf{type}} \qquad \frac{\langle \cdot \rangle\, A^+\ \mathsf{type}}{\langle \Psi \rangle\, \Box A^+\ \mathsf{type}}$$

$$\frac{\begin{array}{l}\vdash_{\text{LF}} \Psi, \Psi'\ \mathsf{ctx} \\ \langle \Psi, \Psi' \rangle\, A^+\ \mathsf{type}\end{array}}{\langle \Psi \rangle\, \Psi \Rightarrow A^+\ \mathsf{type}} \qquad \frac{(\Psi \vdash \Psi' \in \mathcal{W} \ \longrightarrow\ \langle \Psi \rangle\, \psi^+(\Psi')\ \mathsf{type})}{\langle \Psi \rangle\, \exists_{\mathcal{W}}(\psi^+)\ \mathsf{type}}$$

$\boxed{\langle \Psi \rangle\, A^-\ \mathsf{type}}$

$$\frac{\langle \Psi \rangle\, A^+\ \mathsf{type}}{\langle \Psi \rangle \uparrow A^+\ \mathsf{type}} \qquad \frac{\langle \Psi \rangle\, A^+\ \mathsf{type} \quad \langle \Psi \rangle\, B^-\ \mathsf{type}}{\langle \Psi \rangle\, A^+ \to B^-\ \mathsf{type}}$$

$$\frac{}{\langle \Psi \rangle\, \top\ \mathsf{type}} \qquad \frac{\langle \Psi \rangle\, A^-\ \mathsf{type} \quad \langle \Psi \rangle\, B^-\ \mathsf{type}}{\langle \Psi \rangle\, A^- \& B^-\ \mathsf{type}}$$

$$\frac{\Psi \vdash_{\text{LF}} A\ \mathsf{type} \quad (\Psi \vdash_{\text{LF}} M : A \ \longrightarrow\ \langle \Psi \rangle\, \tau^-(M)\ \mathsf{type})}{\langle \Psi \rangle\, \forall_A(\tau^-)\ \mathsf{type}} \qquad \frac{\langle \cdot \rangle\, A^-\ \mathsf{type}}{\langle \Psi \rangle\, \diamond A^-\ \mathsf{type}}$$

$$\frac{\begin{array}{l}\vdash_{\text{LF}} \Psi, \Psi'\ \mathsf{ctx} \\ \langle \Psi, \Psi' \rangle\, A^-\ \mathsf{type}\end{array}}{\langle \Psi \rangle\, \Psi' \curlywedge A^-\ \mathsf{type}} \qquad \frac{(\Psi \vdash \Psi' \in \mathcal{W} \ \longrightarrow\ \langle \Psi \rangle\, \psi^-(\Psi')\ \mathsf{type})}{\langle \Psi \rangle\, \forall_{\mathcal{W}}(\psi^-)\ \mathsf{type}}$$

We write $\langle \Psi \rangle\, A^+\ \mathsf{ok}$ iff $\vdash_{\text{LF}} \Psi\ \mathsf{ctx}$ and $\langle \Psi \rangle\, A^+\ \mathsf{type}$, and similarly for $\langle \Psi \rangle\, A^-\ \mathsf{ok}$. We write $\Delta\ \mathsf{ok}$ iff $\langle \Psi \rangle\, A^-\ \mathsf{ok}$ for all $x : \langle \Psi \rangle\, A^-$ in Δ.

Figure 1: Type formation

Con. pattern	p	::=	$x \mid () \mid (p_1, p_2) \mid \mathsf{inl}\, p \mid \mathsf{inr}\, p$
			$\mid (M, p) \mid \lambda\overline{\Psi}.p \mid \mathsf{box}\, p \mid (\Psi, p)$
Dest. pattern	n	::=	$\epsilon \mid p\,; n \mid \mathsf{fst};\ n \mid \mathsf{snd}; n$
			$\mid M\,; n \mid \mathsf{unpack}\ \overline{\Psi}.n \mid \mathsf{undia}; n \mid \Psi; n$
Context. con. pat.	c	::=	$\overline{\Psi}.p$
Context. dest. pat.	d	::=	$\overline{\Psi}.n$
Context	Δ	::=	$\cdot \mid \Delta, x : C^-$

Figure 2: Constructor and destructor pattern syntax

The remaining positive types are for programming with LF terms. The most basic of these is existential quantification of an LF term, written $\exists_A(\tau^+)$, where A is an LF type, and τ^+ is a meta-function from LF terms M of type A to positive types. We notate meta-functions τ^+ by their graphs—i.e., by a possibly infinite set of non-overlapping pattern branches of the form $M \mapsto A^+$. The formation rule for $\langle\Psi\rangle\, \exists_A(\tau^+)$ requires that A be an LF type in Ψ, and that τ^+ deliver a positive type in Ψ for every LF term in Ψ: we notate iterated inductive definitions by inference rule premises of the form $(\mathcal{J}_1 \longrightarrow \mathcal{J}_2)$. By convention, we tacitly universally quantify over meta-variables that appear first in the premise of an iterated inductive definition, so the second premise of the rule means "for all m, if $\Psi \vdash_{\text{LF}} M : A$ then $\langle\Psi\rangle\, \tau^+(M)\,\mathsf{type}$".

The body of the existential type $\exists_A(\tau^+)$ may be computed from the existentially-quantified LF term in interesting ways. For example, if we define an LF type nat of natural numbers with constructors zero and succ, then we can define a positive type of lists as follows (we may also define it in more traditional ways):

$$
\begin{array}{lcl}
\mathsf{list}\,(A^+) & = & \exists_{\mathsf{nat}}(\tau_{list}) \\
\text{ where} & & \\
\tau_{list}\ \mathsf{zero} & = & 1 \\
\tau_{list}\ (\mathsf{succ}\ \mathsf{zero}) & = & A^+ \\
\tau_{list}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero})) & = & A^+ \otimes A^+ \\
\tau_{list}\ (\mathsf{succ}\ (\mathsf{succ}\ (\mathsf{succ}\ \mathsf{zero}))) & = & A^+ \otimes (A^+ \otimes A^+) \\
 & \vdots &
\end{array}
$$

That is, for every nat n, $\tau_{list}(n)$ is the tuple type $(A^+)^n$. An implementation of our type theory would provide a traditional finitary notation for presenting meta-functions τ^+, e.g., allowing τ_{list} to be defined by recursion.

There are three additional positive types for programming with LF. The types $\Psi \Rightarrow A^+$ and $\Box A^+$ allow for computational language values that manipulate the LF context; their formation rules manipulate the LF context in the same way as their patterns do (see below). Finally, the type $\exists_{\mathcal{W}}(\psi)$ allows existential quantification over the LF contexts in a world $\mathcal{W}$. As with $\exists_A(\tau^+)$, the body of the existential is specified by an abstract pattern-match, this time on LF contexts. This allows types to be defined by computation with LF contexts.

The type formation rules for negative types are analogous. We sometimes abbreviate $\langle\Psi\rangle\, A^+$ by writing C^+ and similarly for C^-.

Operationally, the type formation rules are syntax-directed and well-moded (none

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

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

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

(no rule for 0)

$$\frac{\Delta\,;\Psi \Vdash p :: A^+}{\Delta\,;\Psi \Vdash \mathsf{inl}\, p :: A^+ \oplus B^+} \qquad \frac{\Delta\,;\Psi \Vdash p :: B^+}{\Delta\,;\Psi \Vdash \mathsf{inr}\, p :: A^+ \oplus B^+}$$

$$\frac{\Psi \vdash_{\text{LF}} M : A \quad \Delta\,;\Psi \Vdash p :: \tau^+(M)}{\Delta\,;\Psi \Vdash (M, p) :: \exists_A(\tau^+)} \qquad \frac{\Delta\,;\cdot \Vdash p :: A^+}{\Delta\,;\Psi \Vdash \mathsf{box}\, p :: \Box A^+}$$

$$\frac{\Delta\,;\Psi, \Psi' \Vdash p :: A^+}{\Delta\,;\Psi \Vdash \lambda\overline{\Psi'}.p :: \Psi' \Rightarrow A^+} \qquad \frac{\Psi \vdash \Psi' \in \mathcal{W} \quad \Delta\,;\Psi \Vdash p :: \psi^+(\Psi')}{\Delta\,;\Psi \Vdash (\Psi', p) :: \exists_{\mathcal{W}}(\psi^+)}$$

$\boxed{\Delta\,;\Psi \Vdash n :: A^- > C^+}$

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

$$\frac{\Delta_1\,;\Psi \Vdash p :: A^+ \quad \Delta_2\,;\Psi \Vdash n :: B^- > C^+}{\Delta_1,\Delta_2\,;\Psi \Vdash p\,; n :: A^+ \to B^- > C^+}$$

(no rule for $\top$)

$$\frac{\Delta\,;\Psi \Vdash n :: A^- > C^+}{\Delta\,;\Psi \Vdash \mathsf{fst};\, n :: A^- \& B^- > C^+} \qquad \frac{\Delta\,;\Psi \Vdash n :: B^- > C^+}{\Delta\,;\Psi \Vdash \mathsf{snd}; n :: A^- \& B^- > C^+}$$

$$\frac{\Psi \vdash_{\text{LF}} M : A \quad \Delta\,;\Psi \Vdash n :: \tau^-(M) > C^+}{\Delta\,;\Psi \Vdash M\,; n :: \forall_A(\tau^-) > C^+}$$

$$\frac{\Delta\,;\Psi, \Psi' \Vdash n :: A^- > C^+}{\Delta\,;\Psi \Vdash \mathsf{unpack}\, \overline{\Psi'}.n :: \Psi' \curlywedge A^- > C^+}$$

$$\frac{\Delta\,;\cdot \Vdash n :: A^- > C^+}{\Delta\,;\Psi \Vdash \mathsf{undia}; n :: \diamond A^- > C^+}$$

$$\frac{\Psi \vdash \Psi' \in \mathcal{W} \quad \Delta\,;\Psi \Vdash n :: \psi^-(\Psi') > C^+}{\Delta\,;\Psi \Vdash \Psi'; n :: \forall_{\mathcal{W}}(\psi^-) > C^+}$$

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

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

Figure 3: Constructor and destructor patterns

of the meta-variables appearing in the judgements need to be guessed), with both Ψ and A as inputs. The rules for $\langle\Psi\rangle\ A$ assume and maintain the invariant that $\vdash_{\text{LF}} \Psi\ \mathsf{ctx}$.

2.1.4 Patterns

We present the syntax of patterns in Figure 3 and the rules for pattern formation in Figure 3.

Constructor Patterns Positive types are specified by the judgement $\Delta\,;\Psi \Vdash p :: A^+$, which types *constructor patterns*. This judgement means that p is a constructor pattern for A^+, using the LF variables in Ψ, and binding negative contextual variables $x : \langle\Psi_0\rangle\ A^-_0$ in Δ for all subterms of negative types. The LF variables in Ψ are free in p and A^+ but not Δ: negative assumptions in Δ have no free LF variables, because the free variables of A^- are bound by the context Ψ. Like datatype constructors in ML, constructor patterns are used both to build values and to pattern match. Logically, constructor patterns correspond to using *linear right-rules* to show A^+ from Δ; linearity ensures that a pattern binds a variable exactly once.

The patterns for products and sums are standard. The only pattern for $\downarrow A^-$ is a variable x bound in Δ: one may not pattern-match on negative types such as computational functions. Note that x is bound with a contextual type $\langle\Psi\rangle\ A^-$ capturing the current context Ψ: this contextual type binds the free LF variables of A^+, and ensures that the free LF variables of a term are properly tracked by its type. Moreover, $\downarrow A^-$ is the *only* type at which pattern variables are allowed: patterns may not bind variables at positive types.

Next, we consider the patterns for computing with LF terms. The pattern for $\exists_A(\tau^+)$ is a pair whose first component is an LF term M of type A, and whose second component is a pattern for the positive type $\tau^+(M)$—the type of the second component is computed by applying the meta-function τ^+ to M. For example, returning to the above example of lists defined as $\exists_{\mathsf{nat}}(\tau_{list})$, we have the pattern $(\mathsf{zero}, ())$ representing "nil", because $\tau_{list}(\mathsf{zero}) = 1$. The patterns for $\Psi \Rightarrow A^+$ and $\Box A^+$ manipulate the LF context: $\lambda\overline{\Psi}.p$ binds LF variables (we write $\overline{\Psi}$ for the bare variables of Ψ, without any types), whereas $\mathsf{box}\, p$ wraps a pattern that is independent of the LF context. The pattern for $\exists_{\mathcal{W}}(\psi^+)$ pairs an LF context Ψ with a pattern for the type $\psi(\Psi)$, analogously to $\exists_A(\tau^+)$.

Destructor Patterns Negative connectives are specified by the judgement $\Delta\,;\Psi \Vdash n :: A^- > C^+$, which types *destructor patterns*. A destructor pattern describes the shape of an observation that one can make about a negative type: the judgement means that n observes the negative type A^- to reach the positive conclusion C^+, using the LF variables in Ψ and binding the pattern variables in Δ. The context Ψ scopes over n and A^- but not Δ and C^+—like assumptions, the conclusion C^+, which abbreviates $\langle\Psi_0\rangle\ A^+_0$, is modally encapsulated, potentially in a different context than Ψ. Logically, destructor patterns correspond to using *linear left-rules* to decompose A^- to C^+. Because we are defining an intuitionistic, rather than classical, type theory, destructor patterns are not quite dual to constructor patterns: constructor patterns have no conclusions, whereas destructor patterns have exactly one.

The destructor patterns for the basic types are explained as follows: a negative pair $A^- \& B^-$ can be observed by observing its first component or its second component; negative pairs are lazy pairs whose components are expressions, whereas positive pairs $A^+ \otimes B^+$ are eager pairs of values. A function $A^+ \to B^-$ can be observed by applying it to an argument, represented here by the constructor pattern p, and then observing the result. As a base case, we have shifted positive types $\uparrow A^+$, which represent suspended expressions computing values of type A^+. A suspension can be observed by forcing it, written ϵ, which runs the suspended expression down to a value; the LF context Ψ is encapsulated in the conclusion of the force. The destructor patterns for the remaining types are analogous to their positive counterparts: universal quantification over LF terms $\forall_A(\tau^-)$ is eliminated by choosing an LF term M to apply to, and observing the result; and similarly for universal context quantification. Finally, $\Psi \leftarthreetimes A^-$ and $\diamond A^-$ manipulate the LF context of a negative type.

Contextual Patterns In the focusing framework below, we will require contextually encapsulated patterns with no free LF variables. Contextual constructor patterns c have the form $\overline{\Psi}.p$; they are well-typed when p is well-typed in Ψ. Contextual destructor patterns are similar. In contextual patterns $\overline{\Psi}.p$ and contextual types $\langle \Psi \rangle\ A$, the context Ψ is considered a binding occurrence for all its variables, which may be freely α-converted.

Mode and Regularity The pattern typing rules in Figure 3 are syntax-directed and well-moded: the assumptions Δ and conclusion C^+ of destructor pattern typing, and the assumptions Δ of constructor pattern typing, are outputs (synthesized), whereas all other components of the judgements are inputs. The judgements assume that their inputs are well-formed and guarantee that their outputs are well-formed:

Proposition 1 (Pattern Regularity)**.**

- *If* C^+ ok *and* $\Delta \Vdash c :: C^+$ *then* Δ ok.
- *If* C_0^- ok *and* $\Delta \Vdash d :: C_0^- > C^+$ *then* C^+ ok *and* Δ ok.

2.1.5 Focusing Framework

We present our focusing framework for polarized intuitionistic type theory in Figure 4, which is essentially unchanged from our previous work [42]: the extension with dependent types is localized to the types and their constructor and destructor patterns. In these rules, Γ stands for a sequence of pattern contexts Δ, but Γ itself is treated in an unrestricted manner (i.e., variables are bound once in a pattern, but may be used any number of times within the pattern's scope). As a matter of notation, we regard the diacritic marks on metavariables such as C^+ and C^- as part of the name of the metavariable, not as a modifier, so C^+ and C^- are two unrelated types. The focusing rules are syntax-directed and well-moded, with all pieces of the judgement as inputs.

$$
\begin{array}{llll}
\text{Context} & \Gamma & ::= & \cdot \mid \Gamma, \Delta \\
\text{Pos. Value} & v^+ & ::= & c\,[\sigma] \\
\text{Pos. Cont.} & k^+ & ::= & \epsilon \mid \mathsf{cont}^+(\phi^+) \mid \epsilon \mid k_1^+\ \mathsf{then}_{C^+}\ k_2^+ \\
 & & & \text{where } \phi^+ ::= \{c \mapsto e \mid \cdots\} \\
\text{Expression} & e & ::= & v^+ \mid x \bullet k^- \mid v^- \bullet_{C^-} k^- \mid \mathsf{casev}_{C^+}\ v^+\ \mathsf{of}\ k^+ \mid \mathsf{case}_{C^+}\ e\ \mathsf{of}\ k^+
\end{array}
$$

$$
\begin{array}{llll}
\text{Neg. Cont.} & k^- & ::= & d[\sigma]; k^+ \mid k^-\ \mathsf{then}_{C^+}\ k^+ \\
\text{Neg. Value} & v^- & ::= & x \mid \mathsf{val}^-(\phi^-) \mid x \mid \mathsf{fix}(x.v^-) \\
 & & & \text{where } \phi^- ::= \{d \mapsto e \mid \cdots\} \\
\text{Substitution} & \sigma & ::= & \cdot \mid \sigma, v^-/x \mid \mathsf{id} \mid \sigma_1, \sigma_2
\end{array}
$$

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

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

$\boxed{\Gamma \vdash k^+ : C_0^+ > C^+}$

$$\frac{(\Delta \Vdash c :: C_0^+ \longrightarrow \Gamma, \Delta \vdash \phi^+(c) : C^+)}{\Gamma \vdash \mathsf{cont}^+(\phi^+) : C_0^+ > C^+} \qquad \boxed{\frac{C_0^+ = C^+}{\Gamma \vdash \epsilon : C_0^+ > C^+}} \qquad \boxed{\frac{C_1^+\ \mathsf{ok} \quad \Gamma \vdash k_0^+ : C_0^+ > C_1^+ \quad \Gamma \vdash k_1^+ : C_1^+ > C^+}{\Gamma \vdash k_0^+\ \mathsf{then}_{C_1^+}\ k_1^+ : C_0^+ > C^+}}$$

$\boxed{\Gamma \vdash k^- :: C^- > C^+}$

$$\frac{\Delta \Vdash d :: C^- > C_0^+ \quad \Gamma \vdash \sigma : \Delta \quad \Gamma \vdash k^+ : C_0^+ > C^+}{\Gamma \vdash d[\sigma]; k^+ :: C^- > C^+} \qquad \boxed{\frac{C_0^+\ \mathsf{ok} \quad \Gamma \vdash k^- :: C^- > C_0^+ \quad \Gamma \vdash k^+ : C_0^+ > C^+}{\Gamma \vdash k^-\ \mathsf{then}_{C_0^+}\ k^+ :: C^- > C^+}}$$

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

$$\frac{(\Delta \Vdash d :: C^- > C^+ \longrightarrow \Gamma, \Delta \vdash \phi^-(d) : C^+)}{\Gamma \vdash \mathsf{val}^-(\phi^-) : C^-} \qquad \boxed{\frac{x : C_0^- \in \Gamma \quad C^- = C_0^-}{\Gamma \vdash x : C^-}} \qquad \boxed{\frac{\Gamma, x : C^- \vdash v^- : C^-}{\Gamma \vdash \mathsf{fix}(x.v^-) : C^-}}$$

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

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

$$\boxed{\frac{C^-\ \mathsf{ok} \quad \Gamma \vdash v^- : C^- \quad \Gamma \vdash k^- :: C^- > C^+}{\Gamma \vdash v^- \bullet_{C^-} k^- : C^+}} \qquad \boxed{\frac{C_0^+\ \mathsf{ok} \quad \Gamma \vdash v^+ :: C_0^+ \quad \Gamma \vdash k^+ : C_0^+ > C^+}{\Gamma \vdash \mathsf{casev}_{C_0^+}\ v^+\ \mathsf{of}\ k^+ : C^+}} \qquad \boxed{\frac{C_0^+\ \mathsf{ok} \quad \Gamma \vdash e : C_0^+ \quad \Gamma \vdash k^+ : C_0^+ > C^+}{\Gamma \vdash \mathsf{case}_{C_0^+}\ e\ \mathsf{of}\ k^+ : C^+}}$$

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

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

identity principles | cut principles | convenient principles

Figure 4: Focusing Rules

Canonical Terms First, we discuss canonical terms, which are typed by the unboxed rules in Figure 4. The first two judgements define focusing and inversion for positive types. The judgement $\Gamma \vdash v^+ :: C^+$ defines positive values (right focus): a positive value is a constructor pattern under a substitution for its free variables. The judgement $\Gamma \vdash k^+ : C_0^+ > C^+$ defines positive continuations (left inversion): a positive continuation is a case-analysis, specified by a meta-function ϕ^+ from patterns to expressions. The premise of the rule asserts that for all constructor patterns c for C_0, $\phi^+(c)$ is an expression of the appropriate type using the variables bound by c (by our above convention about iterated inductive definitions, Δ and c are universally quantified here).

The next two judgements define focusing and inversion for the negative types. The judgement $\Gamma \vdash k^- :: C^- > C^+$ defines negative continuations (left focus): a negative continuation is a destructor pattern under a substitution for its free variables followed by a positive continuation consuming the result of the destructor. The destructor pattern, filled in by the substitution, decomposes C^- to some positive type C_0^+. The positive continuation reflects the fact that it may take further case-analysis of C_0^+ to reach the desired conclusion C^+. The judgement $\Gamma \vdash v^- : C^-$ defines negative values (right inversion): a negative value is specified by a meta-function that gives an expression responding to every possible destructor.

The judgement $\Gamma \vdash e : C^+$, types expressions, which are neutral states: from an expression, one can right-focus and introduce a value, or left-focus on an assumption in Γ and apply a negative continuation to it. Finally, a substitution $\Gamma \vdash \sigma : \Delta$ provides a negative value for each hypothesis.

At this point, the reader may wish to work through some instances of these rules (using the above pattern rules) to see that they give the expected typings for familiar types. First, the type $(\uparrow A_1^+)\&(\uparrow A_2^+)$ is inhabited by a lazy pair of expressions:

$$\frac{\Gamma \vdash e_1 : \langle\Psi\rangle\, A_1^+ \quad \Gamma \vdash e_2 : \langle\Psi\rangle\, A_2^+}{\Gamma \vdash \mathsf{val}^-((\overline{\Psi}.(\mathsf{fst};\epsilon)) \mapsto e_1 \mid (\overline{\Psi}.(\mathsf{snd};\epsilon)) \mapsto e_2) : \langle\Psi\rangle\, (\uparrow A_1^+)\&(\uparrow A_2^+)}$$

Second, a function $(\downarrow A_1^-) \oplus (\downarrow A_2^-) \to \uparrow B^+$ is defined by two cases:

$$\frac{\Gamma, x : \langle\Psi\rangle\, A_1^- \vdash e_1 : \langle\Psi\rangle\, B^+ \quad \Gamma, y : \langle\Psi\rangle\, A_2^- \vdash e_2 : \langle\Psi\rangle\, B^+}{\Gamma \vdash \mathsf{val}^-((\overline{\Psi}.\mathsf{inl}\, x) \mapsto e_1 \mid (\overline{\Psi}.\mathsf{inr}\, y) \mapsto e_2) : \langle\Psi\rangle\, (\downarrow A_1^-) \oplus (\downarrow A_2^-) \to \uparrow B^+}$$

In both of these examples, the bindings $\overline{\Psi}$ in the contextual patterns are unused, because there are no LF types mentioned before shifts. As an example where the contextual bindings are relevant, consider an LF type exp representing terms of the untyped λ-calculus. A function from exp to exp is represented by the following negative value:

$$\frac{\Gamma \vdash e_1 : \langle\Psi\rangle\, \exists_{\mathsf{exp}}(\_ \mapsto 1) \quad \ldots}{\Gamma \vdash \mathsf{val}^-((\overline{\Psi}.M_1\,;\epsilon) \mapsto e_1, \ldots) : \langle\Psi\rangle\, \forall_{\mathsf{exp}}(\_ \mapsto \uparrow(\exists_{\mathsf{exp}}(\_ \mapsto 1)))}$$

In a more familiar notation, the type of this term is written $\forall \_ : \mathsf{exp}.\exists \_ : \mathsf{exp}.1$; we assume the meta-functions τ allow constant functions, notated by a catch-all case $\_$. A negative value of this type is given by a meta-function whose domain is destructor patterns for $\langle\Psi\rangle\, \forall_{\mathsf{exp}}(\_ \mapsto \uparrow(\exists_{\mathsf{exp}}(\_ \mapsto 1)))$. All destructor patterns for this type have the

form $\overline{\Psi}.(M_i\,;\epsilon)$ where $\Psi \vdash_{\text{LF}} M : \mathsf{exp}$ because the only destructor pattern for $\forall$ is application to an LF term, and the only destructor pattern for $\uparrow$ is ϵ. Thus, a negative value of this type is specified by an ω-rule with one case for each λ-term in Ψ, and the term M in each pattern is in the scope of the variables bound by $\overline{\Psi}$.

Non-canonical Terms To make a convenient programming language, we add non-canonical forms and general recursion in the boxed rules in Figure 4. The first class of non-canonical forms are internalizations of the cut principles for this presentation of intuitionistic logic; these terms create opportunities for reduction. The most fundamental cuts, $v^- \bullet_{C^+} k^-$ and $\mathsf{casev}_{C^+}\ v^+\ \mathsf{of}\ k^+$, put a value up against a continuation. The three remaining cut principles, $\mathsf{case}_{C^+}\ e\ \mathsf{of}\ k^+$ and $k^-\ \mathsf{then}_{C^+}\ k^+$ and $k_0^+\ \mathsf{then}_{C^+}\ k_1^+$, allow continuations to be composed: the first composes a continuation with an expression, the second composes a negative continuation with a positive one, and the third composes two positive continuations. The second class of non-canonical forms are internalizations of the identity principles, which say that terms need not be fully η-expanded. Negative identity (x) allows a variable to be used as a value, whereas positive identity (ϵ) is the identity case-analysis. The identity substitution (id) maps negative identity across each assumption in Δ. Finally, we allow substitutions to be appended (σ_1, σ_2) so that the identity substitution can be combined with other substitutions, and we allow general-recursive negative values ($\mathsf{fix}(x.v^-)$).

Operational Semantics The operational semantics of our language, defined by the judgement $e \rightsquigarrow e'$ in Figure 5, are quite simple and essentially unchanged from our previous work [42]. Reduction happens when a focus term is put up against the corresponding inversion term. E.g., in the rule pr, a positive value $c\,[\sigma]$ is being scrutinized by a positive continuation $\mathsf{cont}^+(\phi^+)$; this is reduced by applying the meta-function $\phi^+(c)$, which performs the pattern matching, and then applying the substitution σ to the result. Though the types of terms are computationally irrelevant, the operational semantics maintain the annotations on cuts in the interest of a simple type safety result. We elide the definition of substitution ($e\,[\sigma : \Delta]$, and similarly for the other syntactic categories), which is standard, except that it carries the types of the substituted terms so that the substitution into $x \bullet k^-$ can be defined to be $v^- \bullet_{C^-} k^-$ when $v^-/x \in \sigma$ and $x : C^- \in \Delta$.

Type safety is proved by the usual simple structural induction:

Theorem 1 (Type safety)**.**
Progress: *If* $C^+\ \mathsf{ok}$ *and* $\cdot \vdash e : C^+$ *then* $e = v^+$ *or* $e \rightsquigarrow e'$.
Preserv.: *If* $C^+\ \mathsf{ok}$ *and* $\cdot \vdash e : C^+$ *and* $e \rightsquigarrow e'$ *then* $\cdot \vdash e' : C^+$.

2.1.6 Proposed work

Next, we discuss some aspects of this calculus that we will explore further in the thesis.

Type Equality Canonical terms (the unboxed rules in Figure 4) contain no type annotations, and can be checked against a single type annotation provided at the outside. However, non-canonical terms have either too little type information or too much. Cuts

$\boxed{e \rightsquigarrow e'}$

$$\frac{\Delta \Vdash c :: C^+ \quad \phi^+(c)\,\mathsf{defined}}{\mathsf{casev}_{C^+}\ c\,[\sigma]\ \mathsf{of}\ \mathsf{cont}^+(\phi^+) \rightsquigarrow \phi^+(c)\,[\sigma : \Delta]}\ \mathrm{pr}$$

$$\overline{\mathsf{casev}_{C_0^+}\ v^+\ \mathsf{of}\ (k_0^+\ \mathsf{then}_{C_1^+}\ k_1^+) \rightsquigarrow \mathsf{case}_{C_1^+}\ (\mathsf{casev}_{C_0^+}\ v^+\ \mathsf{of}\ k_0^+)\ \mathsf{of}\ k_1^+}$$

$$\frac{}{\mathsf{casev}_{C^+}\ v^+\ \mathsf{of}\ \epsilon \rightsquigarrow v^+}\ \mathrm{idk}^+$$

$$\frac{\Delta \Vdash d :: C_0^- > C^+ \quad \phi^-(d)\,\mathsf{defined}}{\mathsf{val}^-(\phi^-) \bullet_{C_0^-}\ (d[\sigma]; k^+) \rightsquigarrow \mathsf{case}_{C^+}\ (\phi^-(d)\,[\sigma : \Delta])\ \mathsf{of}\ k^+}\ \mathrm{nr}$$

$$\frac{}{v^- \bullet_{C_0^-}\ (k_0^-\ \mathsf{then}_{C_1^+}\ k_1^+) \rightsquigarrow \mathsf{case}_{C_1^+}\ (v^- \bullet_{C_0^-}\ k^-)\ \mathsf{of}\ k^+}\ \mathrm{k}^-\mathrm{k}^+$$

$$\frac{}{\mathsf{fix}(x.v^-) \bullet_{C_0^-}\ k^- \rightsquigarrow v^-\,[(\mathsf{fix}(x.v^-)/x) : (x : C_0^-)] \bullet_{C_0^-}\ k^-}\ \mathrm{fix}$$

$$\frac{e \rightsquigarrow e'}{\mathsf{case}_{C^+}\ e\ \mathsf{of}\ k^+ \rightsquigarrow \mathsf{case}_{C^+}\ e'\ \mathsf{of}\ k^+}\ \mathrm{k}^+\mathrm{ee}$$

$$\frac{}{\mathsf{case}_{C^+}\ v^+\ \mathsf{of}\ k^+ \rightsquigarrow \mathsf{casev}_{C^+}\ v^+\ \mathsf{of}\ k^+}\ \mathrm{k}^+\mathrm{ev}$$

Figure 5: Operational Semantics

have too little type information because they do not obey the subformula property, so we annotate them with the mediating type. On the other hand, identities have too much type information: for example, when x is used as a value, both the type in the context and a type to check against are given. Consequently, type checking identity terms requires comparing two types for equality. Moreover, the identity terms x and ϵ are the *only* terms that force two arbitrary types to be compared for equality, because η-expansion pushes the type equality check down to base type. (For other instances of this phenomenon, see LFR [43], where subtyping at higher types is characterized by an identity coercion, and OTT [2], where an η-expanded identity coercion is induced by proofs of type equality).

In the rules, we write $C_1 = C_2$ for "syntactic" equality of types, which is a congruence with meta-functions compared extensionally—i.e., two meta-functions are equal if they agree on all inputs. However, this notion of equality can lead to undecidability of type checking: extensional equality of the meta-functions appearing in types (τ, ψ) will not in general be decidable. Decidability may be restored in various ways: One option is to restrict τ and ψ to a class of meta-functions whose equality is decidable. For example, if only finite branching without recursion, and not arbitrary type-level computation, is allowed, then equality may be decidable. Alternatively, we may implement a sound but conservative approximation to type equality $C_0 = C$ for use in type checking. When this tactic fails to prove a true equality, the programmer can prove the equality by manually η-expanding the identity coercion (the identity rules are admissible given the other rules of the system). As a practical matter, it may be more convenient to prove equalities explicitly, rather than by η-expanded identity coercions, in which case we could permit explicit equality proofs as part of the identity terms, perhaps by internalizing proofs of type equality as a type in the language. We plan to explore these options in the thesis.

Dependent Pattern Matching Our rules for pattern-matching decompose LF terms and contexts with an infinitary rule, giving one case for each LF term of the appropriate type (e.g., `nat` is pattern-matched with the ω-rule). For example, we illustrate how meta-functions give an abstract account of dependent pattern matching. Consider the `nat` type defined by `zero` and `succ`, with an identity type defined in LF as follows:

```
id   : nat -> nat -> type.
refl : {n : nat} id n n.
```

What are the patterns of type $\exists_{\mathsf{nat}}(n \mapsto \exists_{\mathsf{nat}}(m \mapsto \mathsf{id}\ n\ m))$? In Twelf, one would write `(X , (X , refl X))`, where the unification variable `X` must be used nonlinearly for the pattern to be well-typed. In our formalism, one is required to enumerate all closed instances of this pattern:

$$\begin{array}{l}(\mathsf{zero}, (\mathsf{zero}, \mathsf{refl}\ \mathsf{zero})) \\ (\mathsf{succ}\ \mathsf{zero}, (\mathsf{succ}\ \mathsf{zero}, \mathsf{refl}\ (\mathsf{succ}\ \mathsf{zero}))) \\ \vdots\end{array}$$

Because of the use of meta-functions, the patterns presented above do not include a number of features found in other pattern languages for LF [53, 54, 56]: unification

variables for LF terms, non-linear patterns, unification variables over LF variables, and context variables. These features may play a role in the implementation of meta-functions ϕ, ψ, and τ, which we will explore in the thesis.

Richer Forms of Dependency In the thesis, we will also explore the richer forms of dependency discussed above.

2.2 Effects

A realistic programming language must account for programming with computational effects; in the thesis, we plan to treat computational effects within the focusing formalism described above. As an example, we consider mutable references in this section.

In Figure 6, we present the revised focusing rules. Σ stands for a store typing, with assumptions $l : \mathsf{loc}[C^+]$. There are three new forms of expression: First, one can allocate a new reference cell ($\mathsf{new}\, l : \mathsf{loc}[C^+] := v^+.e$), which binds a variable l standing for the cell, initialized to v^+. Next, one can set the value of a cell ($l := v^+; e$). Finally, one can get ($!l; k^+$) the contents of a cell, pattern-matching the value held there with the positive continuation k^+. In many presentations of mutable references, locations occur only behind-the-scenes in the operational semantics. In this presentation, we do expect programmers to explicitly mention in-scope location variables l in get/set expressions.

In the focus rules ($c\,[\sigma]$ and $d[\sigma]; k^+$), the store typing is carried into the pattern judgements. In Figure 2.2, we add one new type, $\mathsf{ref}\ A^+$, whose pattern packs a location variable as a value of reference type. The inversion rules ($\mathsf{cont}^+(\phi^+)$ and $\mathsf{val}^-(\phi^-)$) quantify over all extensions of the current store typing. This ensures that inversions continue to work after new reference cells have been generated. It also accounts for *pointer equality*: A value of type $\mathsf{ref}\ A^+$ is eliminated by a pattern-match that uncovers the underlying location. A programmer may give cases for each location l_i in scope, testing pointer equality with those known locations. However, because the inversion must work in any future location typing, he must also give a catch-all case for unknown locations. In the remaining rules, the store typing is carried through in the same manner as Γ.

The operational semantics (Figure 8) and type safety proof are straightforward:

Lemma 1 (Store Typing Weakening)**.**

1. *If* $\Sigma\,;\Delta \Vdash \mathcal{J}$ *and* $\Sigma' \geq \Sigma$ *then* $\Sigma'\,;\Delta \Vdash \mathcal{J}$.
2. *If* $\Sigma\,;\Gamma \vdash \mathcal{J}$ *and* $\Sigma' \geq \Sigma$ *then* $\Sigma'\,;\Gamma \vdash \mathcal{J}$.
3. *If* $\Sigma_0 \vdash M : \Sigma$ *and* $\Sigma_0' \geq \Sigma_0$ *then* $\Sigma_0' \vdash M : \Sigma$ *and*

Theorem 2 (Type safety)**.**
Progress: *If* $\Sigma\,;\cdot \vdash e : C^+$ *and* $\Sigma \vdash M : \Sigma$ *then* $e = v^+$ *or there exist* $(M'\,, e')$ *such that* $(M\,, e) \rightsquigarrow (M'\,, e')$.
Preservation: *If* $\Sigma\,;\cdot \vdash e : C^+$ *and* $\Sigma \vdash M : \Sigma$ *and* $(M\,, e) \rightsquigarrow (M'\,, e')$ *then there exists* Σ' *such that* $\Sigma' \geq \Sigma$ *and* $\Sigma' \vdash M' : \Sigma'$ *and* $\Sigma'\,;\cdot \vdash e' : C^+$.

Linear context $\Delta ::= \cdot \mid \Delta, x : C^-$
Unrestricted context $\Gamma ::= \cdot \mid \Gamma, \Delta$
Store typing $\Sigma ::= \cdot \mid \Sigma, l : \mathsf{loc}[C^+]$

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

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

$\boxed{\Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}$

$$\frac{(\Sigma' \geq \Sigma \;\longrightarrow\; (\Sigma'\,;\Delta \Vdash c :: C_0^+ \;\longrightarrow\; \Sigma'\,;\Gamma,\Delta \vdash \phi^+(c) : C^+))}{\Sigma\,;\Gamma \vdash \mathsf{cont}^+(\phi^+) : C_0^+ > C^+}$$

$$\boxed{\frac{C_0^+ = C^+}{\Sigma\,;\Gamma \vdash \epsilon : C_0^+ > C^+}} \qquad \boxed{\frac{\Sigma\,;\Gamma \vdash k_0^+ : C_0^+ > C_1^+ \quad \Sigma\,;\Gamma \vdash k_1^+ : C_1^+ > C^+}{\Sigma\,;\Gamma \vdash k_0^+\,\mathsf{then}_{C_1^+}\,k_1^+ : C_0^+ > C^+}}$$

$\boxed{\Sigma\,;\Gamma \vdash k^- :: C^- > C^+}$

$$\frac{\Sigma\,;\Delta \Vdash d :: C^- > C_0^+ \quad \Sigma\,;\Gamma \vdash \sigma : \Delta \quad \Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}{\Sigma\,;\Gamma \vdash d[\sigma]; k^+ :: C^- > C^+}$$

$$\boxed{\frac{\Sigma\,;\Gamma \vdash k^- :: C^- > C_0^+ \quad \Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}{\Sigma\,;\Gamma \vdash k^-\,\mathsf{then}_{C_0^+}\,k^+ :: C^- > C^+}}$$

$\boxed{\Sigma\,;\Gamma \vdash v^- : C^-}$

$$\frac{(\Sigma' \geq \Sigma \;\longrightarrow\; (\Sigma'\,;\Delta \Vdash d :: C^- > C^+ \;\longrightarrow\; \Sigma'\,;\Gamma,\Delta \vdash \phi^-(d) : C^+))}{\Sigma\,;\Gamma \vdash \mathsf{val}^-(\phi^-) : C^-}$$

$$\boxed{\frac{x : C_0^- \in \Gamma \quad C^- = C_0^-}{\Sigma\,;\Gamma \vdash x : C^-}} \qquad \boxed{\frac{\Sigma\,;\Gamma, x : C^- \vdash v^- : C^-}{\Sigma\,;\Gamma \vdash \mathsf{fix}(x.v^-) : C^-}}$$

$\boxed{\Sigma\,;\Gamma \vdash e : C^+}$

$$\frac{\Sigma\,;\Gamma \vdash v^+ :: C^+}{\Sigma\,;\Gamma \vdash v^+ : C^+} \qquad \frac{x : C^- \in \Gamma \quad \Sigma\,;\Gamma \vdash k^- :: C^- > C^+}{\Sigma\,;\Gamma \vdash x \bullet k^- : C^+}$$

$$\frac{\Sigma\,;\Gamma \vdash v^+ :: C_0^+ \quad \Sigma, l : \mathsf{loc}[C_0^+]\,;\Gamma \vdash e : C^+}{\Sigma\,;\Gamma \vdash \mathsf{new}\, l : \mathsf{loc}[C_0^+] := v^+.e : C^+} \qquad \frac{l : \mathsf{loc}[C_0^+] \in \Sigma \quad \Sigma\,;\Gamma \vdash v^+ :: C_0^+ \quad \Sigma\,;\Gamma \vdash e : C^+}{\Sigma\,;\Gamma \vdash l := v^+; e : C^+}$$

$$\frac{l : \mathsf{loc}[C_0^+] \in \Sigma \quad \Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}{\Sigma\,;\Gamma \vdash !l; k^+ : C^+}$$

$$\boxed{\frac{\Sigma\,;\Gamma \vdash v^- : C^- \quad \Sigma\,;\Gamma \vdash k^- :: C^- > C^+}{\Sigma\,;\Gamma \vdash v^- \bullet_{C^-} k^- : C^+}} \qquad \boxed{\frac{\Sigma\,;\Gamma \vdash v^+ :: C_0^+ \quad \Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}{\Sigma\,;\Gamma \vdash \mathsf{case}v_{C_0^+}\, v^+\, \mathsf{of}\, k^+ : C^+}} \qquad \boxed{\frac{\Sigma\,;\Gamma \vdash e : C_0^+ \quad \Sigma\,;\Gamma \vdash k^+ : C_0^+ > C^+}{\Sigma\,;\Gamma \vdash \mathsf{case}_{C_0^+}\, e\, \mathsf{of}\, k^+ : C^+}}$$

$\boxed{\Sigma\,;\Gamma \vdash \sigma : \Delta}$

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

identity principles | cut principles | convenient principles

Figure 6: Focusing rules with state

$$\frac{l : \mathsf{loc}[A^{+}] \in \Sigma}{\Sigma ; \cdot \Vdash l :: \mathsf{ref}\ A^{+}}$$

Σ flows through the remaining pattern rules as an unrestricted context.

Figure 7: Pattern rules for references

$$\boxed{\Sigma_0 \vdash M : \Sigma}$$

$$\frac{}{\Sigma_0 \vdash \cdot : \cdot} \qquad \frac{\Sigma_0 \vdash M : \Sigma \quad \Sigma_0 \vdash v^{+} :: C^{+}}{\Sigma_0 \vdash M[l :=_{C^{+}} v^{+}] : \Sigma, l : \mathsf{loc}[C^{+}]}$$

$$\boxed{(M\,,\, e) \rightsquigarrow (M'\,,\, e')}$$

$$\frac{\Delta \Vdash c :: C^{+} \quad \phi^{+}(c)\ \mathsf{defined}}{(M\,,\, \mathsf{casev}_{C^{+}}\ c\,[\sigma]\ \mathsf{of}\ \mathsf{cont}^{+}(\phi^{+})) \rightsquigarrow (M\,,\, \phi^{+}(c)\,[\sigma : \Delta])}\ \mathrm{pr}$$

$$\frac{}{(M\,,\, \mathsf{casev}_{C_0^{+}}\ v^{+}\ \mathsf{of}\ (k_0^{+}\ \mathsf{then}_{C_1^{+}}\ k_1^{+})) \rightsquigarrow (M\,,\, \mathsf{case}_{C_1^{+}}\ (\mathsf{casev}_{C_0^{+}}\ v^{+}\ \mathsf{of}\ k_0^{+})\ \mathsf{of}\ k_1^{+})}$$

$$\frac{}{(M\,,\, \mathsf{casev}_{C^{+}}\ v^{+}\ \mathsf{of}\ \epsilon) \rightsquigarrow (M\,,\, v^{+})}\ \mathrm{idk}^{+}$$

- -

$$\frac{\Delta \Vdash d :: C_0^{-} > C^{+} \quad \phi^{-}(d)\ \mathsf{defined}}{(M\,,\, \mathsf{val}^{-}(\phi^{-}) \bullet_{C_0^{-}} (d[\sigma]; k^{+})) \rightsquigarrow (M\,,\, \mathsf{case}_{C^{+}}\ (\phi^{-}(d)\,[\sigma : \Delta])\ \mathsf{of}\ k^{+})}\ \mathrm{nr}$$

$$\frac{}{(M\,,\, v^{-} \bullet_{C_0^{-}} (k_0^{-}\ \mathsf{then}_{C_1^{+}}\ k_1^{+})) \rightsquigarrow (M\,,\, \mathsf{case}_{C_1^{+}}\ (v^{-} \bullet_{C_0^{-}} k^{-})\ \mathsf{of}\ k^{+})}\ \mathrm{k}^{-}\mathrm{k}^{+}$$

$$\frac{}{(M\,,\, \mathsf{fix}(x.v^{-}) \bullet_{C_0^{-}} k^{-}) \rightsquigarrow (M\,,\, v^{-}\,[(\mathsf{fix}(x.v^{-})/x) : (x : C_0^{-})] \bullet_{C_0^{-}} k^{-})}\ \mathrm{fix}$$

- -

$$\frac{(M\,,\, e) \rightsquigarrow (M'\,,\, e')}{(M\,,\, \mathsf{case}_{C^{+}}\ e\ \mathsf{of}\ k^{+}) \rightsquigarrow (M'\,,\, \mathsf{case}_{C^{+}}\ e'\ \mathsf{of}\ k^{+})}\ \mathrm{k}^{+}\mathrm{ee}$$

$$\frac{}{(M\,,\, \mathsf{case}_{C^{+}}\ v^{+}\ \mathsf{of}\ k^{+}) \rightsquigarrow (M\,,\, \mathsf{casev}_{C^{+}}\ v^{+}\ \mathsf{of}\ k^{+})}\ \mathrm{k}^{+}\mathrm{ev}$$

- -

$$\frac{l \# M}{(M\,,\, \mathsf{new}\ l : \mathsf{loc}[C^{+}] := v^{+}.e) \rightsquigarrow (M[l :=_{C^{+}} v^{+}]\,,\, e)}\ \mathrm{new}$$

$$\frac{}{(M[l :=_{C^{+}} \_]\,,\, l := v^{+}; e) \rightsquigarrow (M[l :=_{C^{+}} v^{+}]\,,\, e)}\ \mathrm{set}$$

$$\frac{[l :=_{C^{+}} v^{+}] \in M}{(M\,,\, !l; k^{+}) \rightsquigarrow (M\,,\, \mathsf{casev}_{C^{+}}\ v^{+}\ \mathsf{of}\ k^{+})}\ \mathrm{get}$$

Figure 8: Operational Semantics with references

2.2.1 Proposed Work: Controlling Effects

In the above calculus, storage effects are pervasive, in that any expression of type A^+ may perform storage effects. To integrate effects with dependency, it is useful to make a type distinction between pure and impure computations. In the thesis, we will explore an approach to controlling effects based on indexed polarities—e.g. distinguishing between $\uparrow^{\mathsf{impure}} A^+$, which may have effects, and $\uparrow^{\mathsf{pure}} A^+$, which may not. We conjecture that the Hoare triple type in HTT [48] can be seen as a particularly precise indexed polarity, where the index describes the pre- and post-conditions of the computation. Allowing programmers to define indexed polarities may thus allow HTT to be recovered as a domain-specific logic.

2.3 Proposed Work: Polymorphism and Modularity

Our language for programming with domain-specific logics requires strong support for modularity, both to manage the construction of DSLs themselves and to exploit DSLs in giving rich interfaces to components. In the thesis, we plan to reconsider the type theoretic foundations of modularity (e.g., [21]) from the polarized perspective. There are many interesting issues to address in the design of module systems for dependently typed programming languages, and in particular for languages with LF-like support for variable binding. For example, full dependently typed programming raises issues of information hiding: through dependency, the well-typedness of a piece of code can depend on the *implementation* of a value, not just its type. Consequently, one may not freely replace one implementation of an interface with another. A module system should allow programmers to decide when implementations are are revealed to a module's clients, and to replace one module with another that has equivalent behavior. Another issue is that LF notoriously lacks polymorphism, so datatypes like lists must be replicated for each element type. A treatment of polymorphism and modularity in our setting would address this shortcoming.

3 Practice

The goal of the practical portion of this thesis is to build an implementation that is good enough for testing the type theory by programming with some domain-specific logics. Thus, we expect to produce a basic prototype, not an implementation that is well-engineered enough to be a competitor to, say, SML and Twelf (though we hope this thesis will eventually lead to such an implementation!). In particular, the speed of the type checker and operational semantics will be given only enough attention as is necessary to support some reasonable examples.

There are two main gaps between the above type theory and a practical implementation:

3.1 Proposed Work: Syntax for Meta-functions

The above type theory does not commit to a syntax for meta-functions τ, ψ, ϕ. This has its benefits—we are now free to consider different implementations of meta-functions

without disturbing the meta-theoretic properties of our language—but an implementation must actually provide at least one such syntax. A simple language of meta-functions could consist of two ingredients: First, we may give a syntax of meta-patterns, extending the grammar for constructor patterns c with meta-variables ranging over patterns. A meta-function can then be specified by a finite list of meta-pattern/expression pairs, where the expression is allowed to use meta-variables bound by the pattern to construct values. Type checking these meta-functions will require determining exhaustiveness of patterns (Dunfield and Pientka [23] describe some recent work addressing this problem). Second, we may give a fixed collection of datatype-generic programs witnessing the structural properties of LF (weakening, exchange, contraction, substitution, subordination-based strengthening). This language of meta-functions would allow pattern matching up to a finite depth, which is sufficient for the value level, because we have general recursion in the language. For expressive type-level computation, we may also include recursively defined meta-functions with named auxiliary functions (unless we have already investigated full dependency, in which case the value-level functions could serve this role). More ambitiously, we may investigate ways of making datatype-generic programming available to the programmer.

3.2 Proposed Work: Term Reconstruction

Because dependent types are so precise, they create many opportunities for omitting type and term arguments to functions and using unification to reconstruct them. Indeed, programming without such term reconstruction is usually too tedious to be practical. To achieve a usable implementation with compelling examples, we must implement some form of type and term reconstruction. However, inasmuch as possible, we hope to rely on existing technology [51, 53].

3.3 Examples

Finally, we sketch two simple examples of programming with domain-specific logics. These examples are programmable in the baseline calculus described above. In the thesis, we will consider extensions and more-sophisticated examples that illustrate the additional technology for dependency, effects, and modularity that we develop.

3.3.1 Certified Software: Security-Typed Programming

Security-typed languages, such as Aura [38] and PCML5 [6], use an authorization logic to control access to resources. The basic ingredients of an authorization logic are:

- Resources, such as files and database entries, and principals such as users and programs.
- Atomic propositions describing permissions—e.g., a proposition `K mayread F` for a principal `K` and file resource `F`.
- A modality `K says A` meaning that principal `K` affirms the truth of proposition `A`. The `says` modality permits access control policies to be specified as the ag-

gregation of statements by many different principals, which is important when different principals have jurisdiction over different resources.

Beyond these simple ingredients, there are many choices: Is the logic first-order or higher-order, intuitionistic or classical? What laws should the `says` modality satisfy? How are principals and resources represented? How are principals' statements authenticated? Unlike Aura [38] and PCML5 [6], which provide fixed answers to these questions, our type theory allows programmers to program many different authorization logics, and to combine code written using different logics in a single program.

An Authorization Logic In this section, we define a first-order, intuitionistic authorization logic, where `says` is an indexed lax modality (indexed monad), following Garg and Pfenning [30]. For simplicity, we consider only a fixed collection of principals and resources, represented in LF, and a fixed access control policy. We present an LF encoding of this logic in Figure 9. There are two sorts of terms, `princ`ipals and `res`ources, with a distinguished principal `self` on behalf of whom the programs runs. Propositions include atomic propositions (classified by LF type `aprop`), implication, universal quantification over terms, and the *says* modality `K says A`. The logic is defined as a sequent calculus with one kind of hypothesis (`A hyp`) and two kinds of conclusions: `A true`, and `K affirms A`—the judgement on which the `says` modality is based. We mix prefix, infix, and postfix notation to match the standard syntax for these judgements; note that `|-` binds more loosely than `true` and `affirms`, so `|- A true` is `|- (A true)`. The rules for atomic propositions, implication, and universal quantification are standard, and the rules `aff`, `saysr`, and `saysl` give the return and bind operations for the lax modality. We include `cut` as an explicit rule, for reasons discussed below. This LF encoding uses higher-order abstract syntax to represent the syntax of propositions (e.g., `all`) and to manage the assumptions of the sequent calculus (e.g., all left rules as well as `allr` and `impr` add assumptions to the context; the `alll` rule uses LF function application to perform substitution). Using LF to define logics saves programmers the bureaucracy of implementing variable binding concretely.

In Figure 10, we define principals and resources specific to an application, along with an access control policy for them. As a very simple example, we may control reads to files on a file system. To do so, we define principals for file owners (in this case, `dan`), resources for files (`/home/dan/plan`) and two atomic propositions, stating that a principal owns a resource (written `K owns R`) and that a principal may read a resource (`K mayrd R`). The access-control policy is defined by loading the LF context with certain initial hypotheses; in this case, that Dan owns his plan file (`ownsplan`), that Dan says that all principals may read his plan (`danplan`), and that if the owner of a resource says that some principal can read it, then that principal can read it (`grantrd`). This last axiom provides a controlled way of escaping from the affirmation monad back to truth. Programmers can prove propositions in the logic by constructing LF terms representing derivations; for example, it is simple to show that `self` may read the file `/home/dan/plan` by constructing a derivation of `|- (atom (self mayrd /home/dan/plan)) true`. The derivation uses `danplan`, `ownsplan`, and `grantrd`, as well as logical rules.

```
sort  : type.
princ : sort.
res   : sort.

term  : sort -> type.
self  : term princ.

aprop   : type.
prop    : type.
atom    : aprop -> prop.
implies : prop -> prop -> prop.
says    : term princ -> prop -> prop. %infix says.
all     : (term S -> prop) -> prop.

hyp     : prop -> type.   %postfix hyp.
conc    : type.
true    : prop -> conc.   %postfix true.
affirms : term princ -> prop -> conc. %infix affirms.

|-    : conc -> type.
init  : (atom X) hyp -> |- (atom X) true.
aff   : |- K affirms A
         <- |- A true.
impr  : |- (implies A B) true
         <- (A hyp -> |- B true).
impl  : ((implies A B) hyp -> |- J)
         <- |- A true
         <- (B hyp -> |- J).
saysr : |- (K says A) true
         <- |- K affirms A.
saysl : ((K says A) hyp -> |- K affirms C)
         <- (A hyp -> |- K affirms C).
allr : |- (all ([c] A c)) true
        <- {c : term S} |- (A c) true.
alll : ((all A) hyp -> |- J)
        <- ((A T) hyp -> |- J).
cut  : |- J
       <- |- A true
       <- (A hyp -> |- J).
```

Figure 9: LF Signature for Authorization Logic

```
dan            : term princ.
/home/dan/plan : term res.

owns  : term princ -> term res -> aprop. %infix owns.
mayrd : term princ -> term res -> aprop. %infix mayrd.

ownsplan : (atom (dan owns /home/dan/plan)) hyp.
danplan  : (dan says
             (all [p] atom (p mayrd /home/dan/plan))) hyp.
grantrd : all ([p] (all ([q] (all [r]
           implies (atom (p owns r))
          (implies (p says atom (q mayrd r))
                   (atom (q mayrd r)))))))  hyp.
```

Figure 10: A Policy For File Access

Access-Controlled Operations Now that we have a logic for specifying authorization, we may use it to give rich types to functions that interact with resources, such as a function for reading the contents of a file:

```
read : ⟨ · ⟩ ∀r:term res.
             ∀_:|- (atom (self mayrd r)) true.
              ↑ string
```

To write this type, we use an informal concrete syntax for meta-functions, allowing ourselves to write $\forall X : A.B^+$ for the type $\forall_A(X \mapsto B^+)$ when the meta-function can be defined uniformly with only one pattern branch binding a meta-variable X. To remain in the formalism presented above, we define `string` as $(\exists\_ : \mathsf{lstring}.1)$, where $\mathsf{lstring}$ as an LF type representing lists of characters.

To call this function, a programmer must provide a file resource `r` as well as a proof that the program may read `r`. The resource `r` is used as the file name, and the function returns the contents of the file. The intended invariant of this DSL is that a proof of `self mayrd F` implies that the file `F` exists and that the program has the appropriate file system permissions to read it; if this invariant is violated (i.e. the DSL itself is incorrect), then `read` will abort, e.g. by looping or raising an exception. If a client program uses this interface for all reads, then all reads are authorized by the access control policy. It is important that `read` is typed in the empty LF context (i.e., that its contextual type is $\langle\cdot\rangle\, A^-$): otherwise, clients could simply bind new LF variables standing for proofs and use them to justify a call to `read`.

How is `read` implemented? One option is to simply ignore the proof, map the resource to a string, and call a primitive read function (we did not include I/O effects in the above presentation of our type theory, but they are simple to add). In this case, dependency is used only to enforce an invariant, with no bearing on the actual runtime behavior. Alternatively, following Vaughan et al. [67], we may wish `read` to log the provided proofs for later audit. Administrators can use such logs to diagnose unexpected consequences of an access-control policy. Logging requires a function

```
tostring : ⟨ · ⟩ ∀J:conc. ∀_:(|- J). ↑string
```

which should be implemented by induction on LF terms.

Policy Analysis We can use computation with LF terms to investigate the properties of the stated access control policy. As a very simple example, we may wish to know that the only owner of `/home/dan/plan` is `dan`. We can encode this theorem with a negative value of the following type. Because we included general recursion in the language, a term with this type is not necessarily a proof, but we do not use fix to write this particular term.

$$\begin{array}{lcll} onlydan & : & \langle\cdot\rangle & \forall P : \texttt{term princ}. \\ & & & \forall\_ : \texttt{(atom (P owns /home/dan/plan)) hyp}. \\ & & & \uparrow\exists\_ : \texttt{id P dan}.\, 1 \end{array}$$

This theorem says: for any principal `P` that owns `/home/dan/plan`, `P` is `dan`, where `id` is an LF type family representing equality:

```
id   : term S -> term S -> type.
refl : id T T.
```

We implement $onlydan$ with a meta-function on destructors:

$$onlydan = \mathsf{val}^{-}(\texttt{dan}\,;\texttt{ownsplan}\,;\epsilon \mapsto (\texttt{refl}, ())\,[\cdot])$$

A meta-function ϕ implementing $onlydan$ is well-typed when:

$$(\Delta \Vdash d :: \langle\cdot\rangle\, A^{-} > C^{+} \;\longrightarrow\; \cdot \vdash \phi(d) : C^{+})$$

where A^{-} is the type ascribed to $onlydan$ above. In this LF signature and context, the only destructor pattern of this type is $\texttt{dan}\,;\texttt{ownsplan}\,;\epsilon$, in which case Δ is empty and C^{+} is the contextual type $\langle\cdot\rangle \uparrow\exists\_ : \texttt{id dan dan}.1$—the result type is refined by the case analysis. The positive value $(\texttt{refl}, ())\,[\cdot]$ inhabits this type.

Auditing and Cut Elimination We have deliberately included `cut` as a rule in our authorization logic because the time and space costs of normalizing proofs can be large, and proofs using cut suffice as justifications for `read`. Moreover, logging cut-full proofs may provide clues to auditors [67]. On the other hand, proofs with cut may contain irrelevant detours that make it difficult to see who to blame for unexpected consequences of a policy, whereas the corresponding cut-free proof expresses the direct evidence used to grant access. Thus, it is important to be able to eliminate cuts from log entries during auditing. Fortunately, Garg and Pfenning [30] give a Twelf proof of cut admissibility for their logic, and exploiting open-endedness, we can import their Twelf code as a function in our language.

Let $\mathcal{W}$ stand for LF contexts of the form

$$\texttt{x1:term S1}, \texttt{x2:term S2}, \ldots, \texttt{p1:A1 hyp}, \texttt{p1:A2 hyp}, \ldots$$

for some `Si` and `Aj` (in Twelf, these contexts are described by a *regular worlds* declaration [53]). The key lemma in cut elimination is cut admissibility, which is stated as follows:

$$\begin{array}{ll} \forall & \cdot \vdash \Psi \in \mathcal{W} \\ & \Psi \vdash_{\text{LF}} A : \texttt{prop} \\ & \Psi \vdash_{\text{LF}} C : \texttt{prop} \\ & \Psi \vdash_{\text{LF}} D : \texttt{|-cf A true} \\ & \Psi \vdash_{\text{LF}} D' : \Pi \_ : \texttt{A hyp}.\, \texttt{|-cf C true} : \\ \exists & \Psi \vdash_{\text{LF}} D'' : \texttt{|-cf C true} \end{array}$$

We write `|-cf` for the cut-free version of `|-`, which is specified by all the rules for this judgement in Figure 9 except for `cut`. Cut admissibility proves that one can substitute cut-free evidence for `A` for a hypothesis of `A` and obtain a cut-free result.

The proof of this theorem is a meta-function which can be used to implement a negative value of the following type:

$$\begin{array}{ll} \langle\cdot\rangle\, \forall_{\mathcal{W}}. & \forall A : \texttt{prop}. \\ & \forall C : \texttt{prop}. \\ & \forall D : \texttt{|-cf A true}. \\ & \forall D' : (\Pi \_ : \texttt{A hyp}.\, \texttt{|-cf C true}). \\ & \Uparrow (\exists D'' : \texttt{|-cf C true}.1) \end{array}$$

Here we write $\forall_{\mathcal{W}}.A^{\text{-}}$ for $\forall_{\mathcal{W}}(\Psi \mapsto \Psi \curlywedge A^{\text{-}})$; this type quantifies over all contexts in the world $\mathcal{W}$ and then immediately binds the context in $A^{\text{-}}$. A value of this type is implemented as follows:

$$\mathsf{val}^{\text{-}}(\Psi; \mathsf{unpack}\, \Psi.A\,;\, C\,;\, D\,;\, D'\,;\, \epsilon \mapsto (gp(\Psi, A, C, D, D'), ())\,[\cdot])$$

Inverting the possible destructors for this type yields exactly the premises of the Twelf theorem. To construct a result, we use the notation gp to call Garg and Pfenning's Twelf code to compute an LF term. Twelf is a logic programming language for programming with LF terms, so their proof is not a function but a total relation, which may associate more than one output with each input. We can resolve this non-determinism by simply choosing to return the first result produced by Twelf's proof search.

Alternatively, once we have designed a meta-function language, porting this Twelf proof of cut admissibility will make for a good test case.

3.3.2 Mechanized Metatheory: Logical relations for Gödel's T

Twelf's computational language for proving metatheorems about languages and logics represented in LF permits only $\forall\exists$-statements over LF types. Moving to a higher-order functional programming language like Delphin [56], Belgua [54], and our type theory has a number of advantages. For example, when proving decidability of a judgement $\mathcal{J}$ in Twelf, one must inductively axiomatize its negation $\neg\mathcal{J}$ and prove non-contradiction $(\mathcal{J} \wedge \neg\mathcal{J}) \to 0$ explicitly. With more quantifier complexity, one can define $\neg\mathcal{J}$ as

$\mathcal{J} \to 0$, so non-contradiction is implemented by function application, and prove decidability ($\mathcal{J} \vee (\mathcal{J} \to 0)$).

Additionally, because Twelf allows only $\forall\exists$-statements over LF types, it is not possible to formalize a logical relations argument by interpreting the types of an object language as the types of the Twelf computational language.[1] While Delphin and Beluga have sufficient quantifiers to interpret object-language types, they do not permit the definition of a type by induction on an LF term, which seems necessary to define a logical relation by induction on object-language types. Because our type theory provides type-level computation, we can conduct such logical relations arguments directly, using the quantifiers of our computational language. It is of course possible to formalize this style of argument in a dependent type theory such as Coq or Agda which similarly provides large eliminations; the advantage of our approach is that the programmer can carry out a logical relations argument while using LF to represent the language's binding structure.

As an example, we show how type-level computation with LF terms can be used to type a logical relations argument for the termination of Gödel's T (simply-typed λ-calculus with iteration over natural numbers). For simplicity, we index terms with their types so that only well-typed terms are representable, and we give a call-by-name evaluation relation on closed terms where successor is treated lazily. Binders `lam` and `iter` are represented using higher-order abstract syntax, and the evaluation relation uses LF application to perform substitution.

The ultimate theorem we would like to prove is:

$$\langle\cdot\rangle\, \forall \texttt{A:tp}.\forall \texttt{E:tm A}.\exists \texttt{E':tm A}.\exists \texttt{D:eval E E'}.1$$

The logical relations proof of this theorem works by constructing a closed term model, interpreting the types of Gödel's T as the types of the programming language. The logical relation is defined by induction on object-language types. In our calculus, this is represented by a meta-function ht from LF terms to positive types:

$$\begin{array}{ll}
(\cdot \vdash_{\text{LF}} \texttt{A} : \texttt{tp} \text{ and } \cdot \vdash_{\text{LF}} \texttt{E} : \texttt{tm A} \longrightarrow \langle\cdot\rangle\, ht(\texttt{A}, \texttt{E})\,\mathsf{type}) \\
ht\,\texttt{nat}\,\texttt{E} = & \exists_ : \texttt{htnat E}.1 \\
ht\,\texttt{(arr A1 A2)}\,\texttt{E} = & \exists(\texttt{(}\lambda\ \texttt{u.E')}:\Pi_{:}\texttt{tm A1}.\,\texttt{tm A2}). \\
& \exists \texttt{D:eval E (lam (}\lambda\texttt{u.E'))}. \\
& (\forall \texttt{E1:tm A1}.ht(\texttt{A1}, \texttt{E1}) \to \uparrow ht(\texttt{A2}, \texttt{[E1/u]E'})
\end{array}$$

Here we use one-level pattern-matching and inductive calls to notate the meta-function ht, which maps every Gödel's T type and closed term to a positive type. The case for `arr` says that `E` evaluates to a lambda, and moreover, for every hereditarily terminating argument, the substitution into the body of the lambda is hereditarily terminating. We write `[E1/u]E2` for LF substitution, which is defined as a meta-function on LF terms. The base case refers to an auxiliary relation `htnat` which is defined as follows:

[1] It is possible to formalize logical relations arguments in Twelf by interpreting types as quantifiers in a specification logic encoded in LF [59], but this requires independent verification of the consistency of the specification logic, which is often tantamount to the theorem one is trying to prove.

```
tp  : type.
nat : tp.
arr : tp -> tp -> tp.

tm   : tp -> type.
z    : tm nat.
s    : tm nat -> tm nat.
iter : tm nat -> tm C -> (tm C -> tm C) -> tm C.
lam  : (tm A -> tm B) -> tm (arr A B).
app  : tm (arr A B) -> tm A -> tm B.

eval       : tm A -> tm A -> type.
eval/z     : eval z z.
eval/s     : eval (s E) (s E).
eval/lam   : eval (lam E) (lam E).
eval/iterz : eval (iter E Ez Es) Ez'
              <- eval E z
              <- eval Ez Ez'.
eval/iters : eval (iter E Ez Es) Es'
              <- eval E (s E')
              <- eval (Es (iter E' Ez Es)) Es'.
eval/app   : eval (app E1 E2) E'
              <- eval E1 (lam E)
              <- eval (E E2) E'.
```

Figure 11: LF Representation of Gödel's T

```
htnat   : tm nat -> type.
htnat/z : htnat E
           <- eval E z.
htnat/s : htnat E
           <- eval E (s E')
           <- htnat E'.
```

The fundamental lemma of logical relations states that all well-typed terms are in the relation. One difficultly is that the relation is defined only for closed terms, but for the sake of the proof, the theorem must be generalized to consider open terms. The standard maneuver is to interpret open terms under a grounding substitution of hereditarily terminating terms. To do this, we need a type representing substitutions, which we may define in LF as follows:

```
tplist : type.
tnil   : tplist.
tcons  : tp -> tplist -> tplist.

subst : tplist -> type.
snil  : subst tnil.
scons : tm A -> subst As -> subst (tcons A As).
```

The type `tplist` codes an LF context (u : `tm`, d : `of u A`, ...) by the list (`tcons A ... tnil`). The indexed list `(subst As)` contains one `tm` of type `A` for each `A` in `As`.

We also need a type expressing that a substitution contains hereditarily terminating terms:

$$
\begin{array}{llll}
(\cdot \vdash_{\text{LF}} \mathtt{As} : \mathtt{tplist} \text{ and } \cdot \vdash_{\text{LF}} \mathtt{Es} : \mathtt{subst\ As} & \longrightarrow & \langle\cdot\rangle\, hts(\mathtt{As}, \mathtt{Es})\,\mathsf{type}) \\
hts\ \ \mathtt{tnil} & \mathtt{snil} & = & 1 \\
hts\ \ \mathtt{(tcons\ A\ A2)} & \mathtt{(scons\ E\ Es)} & = & ht(A, E) \otimes hts(As, Es)
\end{array}
$$

Then the fundamental lemma is stated as follows, where $\mathcal{W}$ contain LF contexts (u : `tm`, d : `of u A`, ...).

$$
\begin{array}{l}
\langle\cdot\rangle\, \forall_{\mathcal{W}}(\Psi \mapsto \Psi \curlywedge \forall A : \mathtt{tp}.\forall E : \mathtt{tm\ A}. \\
\qquad \diamond(\forall \mathtt{Es} : \mathtt{subst}(tps\Psi).hts(\mathtt{Es}, (tps\Psi)) \rightarrow \uparrow ht(\mathtt{A}, \mathtt{E\ [Es]}))))
\end{array}
$$

For any Ψ in $\mathcal{W}$, given an `E` of type `A` in Ψ, along with a closed hereditarily terminating substitution `Es` for each of the free variables of `E`, we produce a proof that the simultaneous substitution `E [Es]` is hereditarily terminating. The type $\diamond$ is used to express the fact that the substitution consists of closed terms. The meta-operation $tps\Psi$, codes a context Ψ as a `tplist`; it is defined by induction on Ψ. The meta-function `E [Es]` implements simultaneous substitution for LF terms. This meta-function need not be implemented directly for this instance: it can be derived from a generic simultaneous substitution theorem for LF.

We implement this type by induction on `E`, using standard lemmas (closure under head expansion, and an inductive lemma showing that the iterator is in the relation). The proof uses several extensional type equalities involving properties of simultaneous substitution. These equalities are true (e.g., they were proved by Harper and Pfenning

[33] in the course of studying LF using logical relations), and because we treat equality extensionally, they are not reflected in the proof term. We plan to study a concrete language for type equality proofs in the thesis.

We consider a fully fleshed-out version of this logical relations proof to be an interesting test case for our implementation.

4 Related Work

The work described in this thesis will make two technical contributions relative to prior work: First, we will investigate issues such as variable binding, dependency, effects, and modules from from the perspective of polarized type theory—a mathematically interesting endeavor that has already led to new insights, such as our integration of variable binding and computation in previous work [42]. Polarity [31, 32] and focusing [4] are logical ideas whose type-theoretic applications have just begun to be explored. For example, they can be used to explain evaluation order [40, 72], and they draw out the duality between proofs and refutations in computational interpretations of classical logic (see, for example, Curien and Herbelin [19], Filinski [24], Selinger [61], Zeilberger [72]).

Second, the language we arrive at will provide better support for programming with domain-specific logics than existing languages do. At this point, we can justify this claim by contrasting our integrated approach to binding and computation [42] with the way one programs with variable binding in these existing languages.

Dependently Typed Programming Languages There has been a great deal of work on integrating various forms of dependent types into practical programming languages and their implementations [5, 13, 14, 15, 16, 22, 26, 27, 45, 48, 51, 52, 62, 63, 65, 69, 70, 71, 74], building on dependently typed proof assistants such as NuPRL [17] and Coq [9]. However, none of these languages provide built-in support for representing variable binding and hypothetical judgements, which are essential ingredients of domain-specific logics. Thus, programmers must implement variable binding on a case-by-case basis, using one of the following techniques.

Variable binding can be implemented in a variety of ways (see Aydemir et al. [7] for a survey). Among the concrete representation techniques, our approach is most similar to representations where the context of a term is marked in its type, such as de Bruijn representations using nested types or dependency [1, 8, 10]. In these representations, binders introduce a new constructor for variables, which are explicitly injected into terms. Our framework builds this use of dependency into the language: all types are contextual and all datatypes may be extended by rule variables introduced by logical connectives. This creates an opportunity to implement structural properties once for all types, including negative types such as computational functions, and to abstract away from the concrete implementation of variables themselves—as in LF, we can provide a named notation without requiring the programmer to manage names.

Alternatively, it is tempting to try to reuse the computational function space of a functional programming language to represent binding, but the naive approach admits too many functions. One solution to this problem is to use a predicate to identify

those computational functions that are in fact substitution functions [3, 12, 20, 35, 47]. Another solution is to bind meta-language variables of an abstract type defined only by an axiomatic characterization of the properties of variables [11]. In contrast, our representational functions provide a direct means of adequately encoding binding, without requiring side conditions or axioms.

Twelf, Delphin, Beluga In systems based on the LF logical framework [34], LF is taken as a pure representation language, and a separate layer is provided for computation. In Twelf [53], LF/ML [58] Delphin [56], and Beluga [54], the computational layer is an entirely separate language. Schürmann et al. [60] describe an approach in which the same arrow is used for both computation and representation, with primitive recursion isolated by a modality, but computation is nonetheless segregated because the computational modality cannot appear in rules. These stratified approaches have the advantage that all representations automatically obey the structural properties of a hypothetical judgement, with the disadvantage that certain encoding techniques, which rely on embedding computation in data, are not possible. Our approach removes this stratification, allowing rules that embed computation, with the consequence that not all representable rule systems satisfy the structural properties. However, we may implement strengthening, weakening, and substitution generically under certain subordination conditions. Consequently, our framework provides meta-operations implementing the structural properties "for free" for all rule systems definable in simply-typed LF, as well as for many more rule systems that use iterated inductive definitions. Another contribution relative to existing LF-based approaches is that we provide an account of type-level computation with LF terms, as illustrated in the logical-relations example above.

Technically, our contextual modality $\langle \Psi \rangle\, A$ is different than that of contextual modal type theory [49] and Beluga, where contextual variables are eliminated by substitution. Because side conditions, expressed as computational functions, can invalidate structural properties such as weakening and substitution, and thus the type theory must not commit to these properties by building them into the meaning of contextual types. Instead of eliminating contextual types by substitution, we allow pattern matching on contextual types, and view substitution as an admissible property (when it is true!), defined in the meta-function language.

Our contextual types are also related to those in FO$\lambda^{\Delta\nabla}$ [46]. Miller and Tiu's self-dual ∇ connective is closely related to $\Rightarrow$ and λ, also capturing the notion of a scoped constant. An essential difference, however, is that because the ∇ proof theory adopts a logic programming-based distinction between propositions and types (∇ quantifies over a type and forms a proposition), it is significantly less subtle than our work. For example, ∇ cannot appear in the domain of a ∇ (in contrast to $\Rightarrow$).

Nominal Logic Nominal logic [28] is a theory of names and binding that has been implemented in several programming languages (e.g., FreshML [55, 64] and the Isabelle nominal datatype package [66]). The differences between the nominal approach and ours stem from the fact that FreshML separates fresh name generation from the binding of a name in a scope, whereas in our type theory rule variables do not ex-

ist outside of the scope in which they are bound. Nominal logic facilitates the direct representation of informal algorithms that use names without being explicit about their scope, whereas our approach follows the LF methodology of recasting these algorithms in terms of a more disciplined binding structure. Separating name generation from scoping makes it more difficult to determine what names are free in a computation, requiring freshness analyses [55], specification logics [57], or stateful operational semantics [64] in order to ensure that functions respect α-equivalence of representations. In contrast, the free rule variables of all computations are tracked by our type system, and respect for α-equivalence is achieved simply, by quotienting patterns by α-equivalence.

Semantics Fiore et al. [25] and Hofmann [36] give semantic accounts of variable binding. It would be interesting to see whether these semantic accounts can be extended to rule systems which permit computational functions in premises.

5 Plan

In summary, I will substantiate my thesis statement by

- Designing a polarized type theory for programming with domain-specific logics, with support for dependent types, computational effects, and polymorphism.
- Implementing that type theory and programming several examples.

The time spent on the thesis will be apportioned as follows:

40% Theory

- 20% Dependency
- 5% Effects
- 15% Polymorphism and modularity

40% Practice

- 10% Meta-function language
- 10% Term reconstruction
- 20% Implementation and examples

20% Writing

Acknowledgements

This thesis proposal is based on research papers authored jointly with Noam Zeilberger and Robert Harper.

References

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

[2] T. Altenkirch, C. McBride, and W. Swierstra. Observational equality, now! In *Programming Languages meets Program Verification Workshop*, 2007.

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

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

[5] L. Augustsson. Cayenne - a language with dependent types. In *International Conference on Functional Programming*, 1998.

[6] K. Avijit and R. Harper. A language for access control. Technical Report CMU-CS-07-140, Carnegie Mellon University, Computer Science Department, 2007.

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

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

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

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

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

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

[13] C. Chen and H. Xi. Combining programming with theorem proving. In *International Conference on Functional Programming*, 2005.

[14] J. Cheney and R. Hinze. Phantom types. Technical Report CUCIS TR20003-1901, Cornell University, 2003.

[15] B. Chin, S. Markstrum, and T. Millstein. Semantic type qualifiers. In *Programming Language Design and Implementation*, 2005.

[16] J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

[17] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice Hall, 1986.

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

[19] P.-L. Curien and H. Herbelin. The duality of computation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 233–243, 2000.

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

[21] D. Dreyer, K. Crary, and R. Harper. A type theory for higher-order modules. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

[22] J. Dunfield and F. Pfenning. Tridirectional typechecking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.

[23] J. Dunfield and B. Pientka. Case analysis on higher-order data. Draft: `http://www.cs.mcgill.ca/~complogic/beluga/`, February 2008.

[24] A. Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, 1989. Computer Science Department.

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

[26] C. Flanagan. Hybrid type checking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 245–256, 2006.

[27] S. Fogarty, E. Pasalic, J. Siek, and W. Taha. Concoqtion: indexed types now! In *ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, pages 112–121, New York, NY, USA, 2007. ACM Press. ISBN 978-1-59593-620-2.

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

[29] I. Galois. Cryptol reference manual, 2002.

[30] D. Garg and F. Pfenning. Non-interference in constructive authorization logic. In *Proceedings of the 19th IEEE Computer Security Foundations Workshop (CSFW 19)*, 2006.

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

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

[33] R. Harper and F. Pfenning. On equivalence and canonical forms in the LF type theory. *ACM Transactions on Computational Logic*, 6:61–101, 2005.

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

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

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

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

[38] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, , and S. Zdancewic. Aura: A programming language for authorization and audit. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

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

[40] P. B. Levy. *Call-by-push-value*. PhD thesis, Queen Mary, University of London, 2001.

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

[42] D. R. Licata, N. Zeilberger, and R. Harper. Focusing on binding and computation. In *IEEE Symposium on Logic in Computer Science*, 2008.

[43] W. Lovas and F. Pfenning. A bidirectional refinement type system for LF. *Electronic Notes in Theoretical Computer Science*, 196:113–128, 2008.

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

[45] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 15(1), 2004.

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

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

[48] A. Nanevski, G. Morrisett, and L. Birkedal. Polymorphism and separation in Hoare Type Theory. In *ACM SIGPLAN International Conference on Functional Programming*, pages 62–73, Portland, Oregon, 2006.

[49] A. Nanevski, F. Pfenning, and B. Pientka. Contextual modal type theory. *Transactions on*

Computational Logic, 2007. To appear.

[50] A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

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

[52] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, 2006.

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

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

[55] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

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

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

[58] S. Sarkar. A cost-effective foundational certified code system. Thesis Proposal, Carenegie Mellon University, 2005.

[59] J. Sarnat and C. Schürmann. Structural logical relations. In *IEEE Symposium on Logic in Computer Science*, 2008.

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

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

[62] Z. Shao, V. Trifonov, B. Saha, and N. Papaspyrou. A type system for certified binaries. *ACM Transactions on Programming Languages and Systems*, 27(1):1–45, 2005.

[63] T. Sheard. Languages of the future. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.

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

[65] M. Sozeau. PROGRAM-ing finger trees in Coq. In *ACM SIGPLAN International Conference on Functional Programming*. Association for Computing Machinery, 2007.

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

[67] J. A. Vaughan, L. Jia, K. Mazurak, and S. Zdancewic. Evidence-based audit. In *Proceedings of the 21st IEEE Computer Security Foundations Symposium*, Pittsburgh, PA, USA, June 2008.

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

[69] E. Westbrook, A. Stump, and I. Wehrman. A language-based approach to functionally correct imperative programming. In *International Conference on Functional Programming*, 2005.

[70] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Conference on Programming Language Design and Implementation*, 1998.

[71] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2003.

[72] N. Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1–3), 2008.

Special issue on “Classical Logic and Computation”.

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

[74] C. Zenger. *Indizierte Typen*. PhD thesis, Universit at Karlsruhe, 1998.