Dependently Typed Programming with Domain-Specific Logics

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

Abstract

We define a dependent programming language in which programmers can define and compute with *domain-specific logics*, such as an access-control logic that statically prevents unauthorized access to controlled resources. Our language permits programmers to define logics using the LF logical framework, whose notion of binding and scope facilitates the representation of the consequence relation of a logic, and to compute with logics by writing functional programs over LF terms. These functional programs can be used to compute values at run-time, and also to compute types at compile-time. In previous work, we studied a simply-typed framework for representing and computing with variable binding [LICS 2008]. In this paper, we generalize our previous type theory to account for dependently typed inference rules, which are necessary to adequately represent domain-specific logics, and we present examples of using our type theory for certified software and mechanized metatheory.

1. Introduction

In this paper, we describe a functional language for programming with *domain-specific logical systems*—i.e., new programming languages and logics relevant to a particular programming task. Applications of domain-specific logics include both *mechanized metatheory,* where studying a logical system is itself the goal, and *certified software*, where a domain-specific logic is used to establish properties of run-time code. Examples of the latter include:

- Cryptol [22], a language for implementing cryptographic protocols, tracks the lengths of vectors statically, in the style of DML [60].
- Security-typed programming languages such as Aura [30] and PCML5 [4] employ authorization logics to statically prevent unauthorized access to controlled resources.
- Ynot [41], an implementation of Hoare Type Theory [39], provides a separation logic for reasoning about imperative code.

These languages' type systems are domain-specific logics for reasoning about a particular programming style or application domain. Some logics may admit effective decision procedure (e.g., linear arithmetic), while others require non-trivial proofs (e.g., separation logic). The goal of the present work is to give a single host language in which domain-specific logics like these may be embedded, so that programmers may easily define domain-specific logics, reason about them, and use them to reason about code. With such a host language, new logics may be implemented as libraries, not new languages.

To achieve this goal, a host language must provide the means to represent and compute with logical systems. Both of these tasks can be accomplished in a dependently typed functional programming language: dependent types are sufficiently rich to adequately represent the deductive apparatus of logical systems, and functional programs allow computation by structural recursion on syntax and derivations. Indeed, dependent types have already proved quite useful for these tasks—e.g., for mechanizing metatheory in Coq [13], or for embedding typed domain-specific languages in Agda [43].

However, it is our thesis that significant applications of domain-specific logics require good support for representing and computing with binding and scope—i.e., 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. Whereas languages such as Agda and Coq provide no intrinsic support for binding and scope, the LF logical framework [27], itself a dependently typed λ-calculus, supports facile representations of binding using the LF function space. 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 [45], Delphin [48], and Beluga [46]. In previous work [34], we investigated an alternative approach, using the logical notions of polarity [25] and focusing [2] to integrate representational and computational functions as two types in a single, simply-typed, logical framework. This integrated approach permits inference rules that mix iterated inductive definitions [36] and hypothetical judgements. However, adequately representing domain-specific logics necessitates a dependently typed framework, and we do not wish to pursue dependency on computation at this time. Thus, we adopt a stratified approach here, taking LF as a separate representation language, and leaving richer dependency to future work.

In this paper, we adapt our previous techniques for computing with binding and scope to a dependently typed framework, yielding a language suitable for programming with domain-specific logics:

1. We define and study a type theory in which programmers may define domain-specific languages and logics using LF, compute with LF terms via pattern-matching functional programs, and define types by recursion on LF terms. Type-level computation is not provided by any previous computational language for LF.
2. We demonstrate the expressiveness of this framework by giving examples of certified software and mechanized metatheory: we embed a security-typed language in the style of Aura and PCML5, and we show the role of type-level computation in formalizing a logical relations argument.

Our type theory is organized around the logical notions of polarity [25] and focused proofs [2], exploiting the Curry-Howard correspondence between focused proofs and pattern-matching functional programs. As in our previous joint work, we follow Zeilberger's higher-order formulation of focusing [34, 62, 63]:

- The syntax of programs reflects the interplay of *focus* (choosing patterns) and *inversion* (pattern matching), with individual types defined by their pattern typing rules.
- The syntax of types is *polarized*, distinguishing positive data (introduced by focus, eliminated by inversion) from negative computation (introduced by inversion, eliminated by focus).
- 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 [36].

While this style of presentation may be unfamiliar to some readers, it has several advantages: First, polarized types provides a natural framework for integrating representations of logics (as positive types) and computation with them (as negative types). Second, the use of meta-functions to represent pattern-matching allows our type theory to be computationally open-ended (cf. Howe [29]) 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. Abstracting the syntax for pattern matching out of the core type theory affords the freedom to use several different notations for pattern matching in a single program, and to import functions from other languages and systems. For example, as we show below, we may import all existing Twelf functions as inhabitants of certain types in our language. Third, focused proofs emphasize pattern matching as the means of computing with positive data; this naturally extends to the definition of types by pattern-matching on positive data.

In addition to polarity and focusing, our type theory makes essential use of contextual types, inspired by Contextual Modal Type Theory [40] and FO$\lambda^{\Delta\nabla}$ [38], to manage the scoping of LF variables. We take a *pronominal* approach to variables: every variable occurrence is a reference to a binding site, either in a term or in a context. This is different than approaches based on nominal logic [54], where names exist independently of a scope; consequently, we avoid stateful name generation and can easily scale to dependency on syntax with binding. However, there are technical differences between our treatment of contextual types and those in previous work, as we discuss further below.

Organization We describe our type theory in Section 2, examples in Section 3, and related work in Section 4.

2. A Type Theory for Domain-Specific Logics

Our 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 2).
  - A focusing framework (Figure 3) and its operational semantics (Figure 4)

We discuss LF in Section 2.1, types and patterns in Section 2.2, and the focusing framework in Section 2.3.

2.1 LF

We briefly review the LF methodology for representing languages and logics [27]: 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 [58]:

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. [58] discuss type formation and typing; one possible definition of worlds $\mathcal{W}$ is the regular worlds notation of Twelf [45].

2.2 Types and Patterns

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 [2, 24, 31, 33, 62] 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 [34, 62, 63], types are specified by patterns, which are used in both focus and inversion: focus phases choose a pattern, whereas inversion phases pattern-match. In

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

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

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

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

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

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

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

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

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

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

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

We write $\langle \Psi \rangle\ A^+\ \mathsf{ok}$ iff $\vdash_{\mathrm{LF}}\ \Psi\ \mathsf{ctx}$ 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

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 2.3, which uses an iterated inductive definition quantifying over them to specify inversion.

2.2.1 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.

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

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

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

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

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

(no rule for 0)

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

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

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

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

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

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

(no rule for $\top$)

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

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

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

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

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

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

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

Figure 2. Constructor and destructor patterns

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 $\Psi \vdash_{\mathrm{LF}} M : A$ 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 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 $\vdash_{\mathrm{LF}} \Psi\ \mathsf{ctx}$.

2.2.2 Patterns

We present the rules for pattern formation in Figure 2.

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, $\Psi \curlywedge A^-$ 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 2 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* $C^+\ \mathsf{ok}$ *and* $\Delta \Vdash c :: C^+$ *then* $\Delta\ \mathsf{ok}$.
- *If* $C^-_0\ \mathsf{ok}$ *and* $\Delta \Vdash d :: C^-_0 > C^+$ *then* $C^+\ \mathsf{ok}$ *and* $\Delta\ \mathsf{ok}$.

2.3 Focusing Framework

We present our focusing framework for polarized intuitionistic type theory in Figure 3, which is essentially unchanged from our previous work [34]: 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

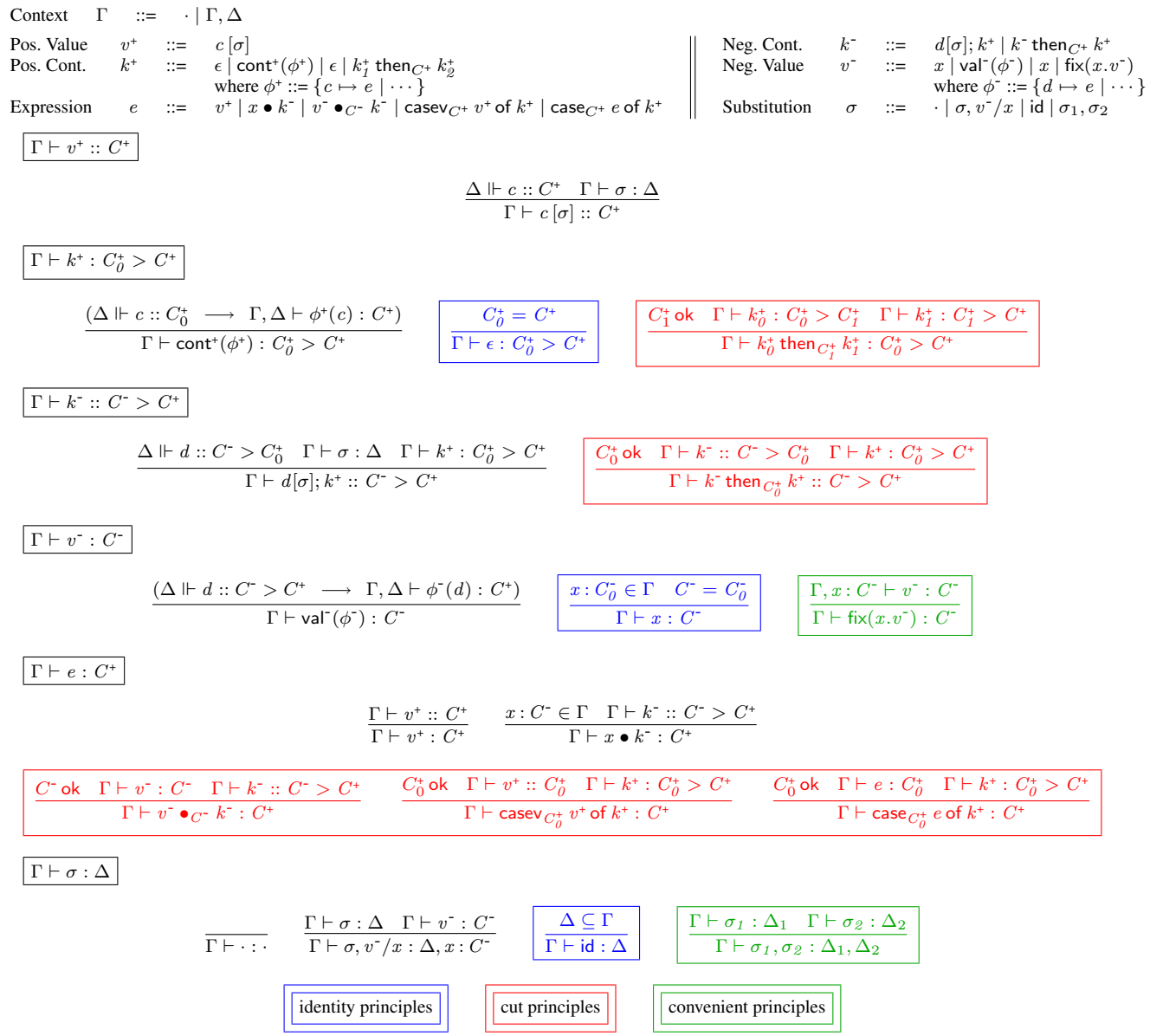

Figure 3. Focusing rules

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.

Canonical Terms First, we discuss canonical terms, which are typed by the unboxed rules in Figure 3. 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 $\Psi \vdash_{\mathrm{LF}} M : \mathsf{exp}$ 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 3. 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 ($\mathsf{fix}(x.v^-)$).

Canonical terms (the unboxed rules in Figure 3) 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 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 [35], where subtyping at higher types is characterized by an identity coercion, and OTT [1], 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 straightforward congruence with meta-functions compared extensionally—i.e., two meta-functions are equal if they agree on all inputs.

$\boxed{e \rightsquigarrow e'}$

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

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

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

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

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

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

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

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

Figure 4. Operational Semantics

Operational Semantics The operational semantics of our language, defined by the judgement $e \rightsquigarrow e'$ in Figure 4, are quite simple and essentially unchanged from our previous work [34]. 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* $\cdot \vdash e' : C^+$.

Decidability of Type Checking Because all of the judgements of our type theory are syntax-directed and well-moded, a simple induction reduces their decidability to decidability of meta-function typing and equality:

Theorem 2 (Decidability)**.** *Realize meta-functions τ, ψ with an implementation that admits decidable type checking and equality, and realize meta-functions ϕ with an implementation that admits decidable type checking. Then all type formation, pattern formation, and focusing framework judgements are decidable.*

It is reasonable to assume an effective procedure for type checking an implementation of meta-functions—e.g., if the meta-functions are presented by a finite set of branches with positive pattern variables standing for unexplored parts of a pattern, then one need only type check a finite number of cases. However, extensional equality of the meta-functions appearing in types (τ, ψ) will not in general be decidable. Nonetheless, decidability can be restored in various ways: One option is to restrict τ and ψ to a class of meta-functions whose equality is decidable. E.g. 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 future work.

2.4 Discussion

Now that we have given a technical presentation of our type theory, we call attention to some subtle aspects of our treatment of computation with LF terms.

α-equivalence Positive continuations k^+ and negative values v^- are defined using meta-functions ϕ on patterns, which contain LF terms. We ensure that these meta-functions respect α-equivalence of LF terms by defining them on α-equivalence classes of patterns, where the definition of α-equivalence for patterns is a straightforward extension of the definition of α-equivalence for LF: The context Ψ in the judgements $\Delta\,;\Psi \Vdash p :: A^+$ and $\Delta\,;\Psi \Vdash n :: A^- > C^+$ binds LF variables, as do the binding forms $\overline{\Psi}.p$, $\overline{\Psi}.n$, $\lambda\overline{\Psi}.p$, and $\mathsf{unpack}\ \overline{\Psi}.d$, and the contexts Ψ in contextual assumptions and conclusions $\langle\Psi\rangle\ A^+$ and $\langle\Psi\rangle\ A^-$. All of these binders can be independently α-renamed.

Scoping We support a variety of types that manipulate the LF context ($\Rightarrow$/$\mathbin{\curlywedge}$,$\Box$/$\Diamond$) by associating an LF context with each assumption and conclusion in a sequent $C_1^-, \ldots, C_n^- \vdash e : C^+$. This results in many different LF contexts scoping over different parts of a computation. For example, a positive continuation k^+ is typed by two LF contexts, one for the input to the continuation (giving the LF variables that may be used in patterns), and another for the output (giving the LF variables that may be used in the result)—a continuation may consume terms in one context and produce terms in another. A positive value v^+ has one LF context determining the variables in scope in the pattern, and other, potentially different, LF contexts for each negative value in the substitution, allowing for patterns that bind LF variables in a negative subterm. The type annotations on cuts also cause a context switch. E.g., in $\mathsf{case}v_{\langle\Psi\rangle A^+}\ v^+\ \mathsf{of}\ k^+$, the value v^+ and the patterns of the continuation k^+ are in the LF context Ψ, but the conclusion of k^+ may be in a different context.

Adequacy The LF methodology relies on bijections between the syntax and derivations of a DSL and the LF terms of particular types. We may import these adequacy results into our language because the positive values of type $\exists_A(\_ \mapsto 1)$ are essentially the LF terms of type A (if we canonized substitutions σ by treating id and σ_1, σ_2 as derived forms, then this would be a bijection):

Proposition 2 (Adequacy)**.**

- *If* $\Psi \vdash_{LF} M : A$ *then* $\Gamma \vdash (\overline{\Psi}.(M, ()))\,[\cdot] :: \langle\Psi\rangle\ \exists_A(\_ \mapsto 1)$.
- *If* $\Gamma \vdash v^+ :: \langle\Psi\rangle\ \exists_A(\_ \mapsto 1)$*, then* v^+ *is* $(\overline{\Psi}.(M, ()))\,[\sigma]$ *where* $\Gamma \vdash \sigma : \cdot$ *and* $\Psi \vdash_{LF} M : A$.

Thus, by type safety, we know that any closed expression of type $\langle\Psi\rangle\ \exists_A(\_ \mapsto 1)$ either is an LF term of type A or steps towards one.

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). Consequently, the patterns presented above do not include a number of features found in other pattern languages for LF [45, 46, 48]: 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 do not specify.

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 non-linearly 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 we allow dependency only on LF terms, dependency cannot force negative variables from Δ to be used non-linearly (though dependency on computation would do so).

3. Examples

3.1 Security-Typed Programming

Security-typed languages, such as Aura [30] and PCML5 [4], 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 aggregation 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 [30] and PCML5 [4], 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.

```
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).

%% a policy for file access:

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 5. LF Signature for Authorization Logic

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 [23]. 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 5. 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.

Next, 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.

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 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 run-time behavior. Alternatively, following Vaughan et al. [57], 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 can 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.

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

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:

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

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 `dan ; ownsplan ; ϵ`, 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 [57]. 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 [23] 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

`x1:term S1, x2:term S2, ..., p1:A1 hyp, p1:A2 hyp, ...`

for some `Si` and `Aj` (in Twelf, these contexts are described by a *regular worlds* declaration [45]). The key lemma in cut elimination is cut admissibility, which is stated as follows:

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

We write `|-cf` for the cut-free version of `|-`, which is specified by all the rules for this judgement in Figure 5 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:

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

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

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

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.

Discussion We hope to have suggested with the above example that our type theory has an appropriate type structure for embedding a security-typed language. However, the above security-typed language is quite limited in several ways: it relies on a static collection of principals, resources, and policies; atomic propositions such as `mayrd` can only refer to LF terms; it only allows execution on behalf of one principal (`self`). We plan to consider embedding more-extensive security-typed languages in future work.

3.2 Logical relations for Gödel's T

Twelf's computational language for proving metatheorems about languages and logics represented in LF permits only ∀∃-statements over LF types. Moving to a higher-order functional programming language like Delphin [48], Belgua [46], 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 ∀∃-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.

[1] It is possible to formalize logical relations arguments in Twelf by interpreting types as quantifiers in a specification logic encoded in LF [50], 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 6. LF Representation of Gödel's T

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:

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

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:

```
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:

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

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

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

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 [26] 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 future work.

4. Related Work

There has been a great deal of work on integrating various forms of dependent types into practical programming languages and their implementations [3, 7, 9, 10, 11, 15, 19, 20, 37, 39, 42, 44, 52, 53, 55, 59, 60, 61, 64], building on dependently typed proof assistants such as NuPRL [12] and Coq [6]. However, none of these languages provide built-in support for representing variable binding and hypothetical judgements, which are essential ingredients of domain-specific logics.

In contrast, our language builds on a wide range of experience representing logical systems in LF [27] and computing with them in Twelf [45], and thus is most closely related to the functional languages LF/ML [49], Delphin [48], and Beluga [46]. LF/ML allows run-time datatypes dependent on LF terms, which permits e.g. writing a type checker that returns an LF certificate that a program is well-typed. However, LF/ML does not allow pattern-matching computation with LF terms themselves, either at the type level or at the value level. Relative to Delphin and Beluga, our contribution is to provide an account of type-level computation with LF

terms, and to investigate a different formalism for integrating open LF terms into a computational language. For example, in Delphin, all types are interpreted relative to one ambient LF context, as in Twelf. With this type structure, it is unclear how one would express functions that take different arguments in different contexts; e.g., the type of the fundamental lemma in Section 3.2 uses our connective $\diamond$ to express a substitution containing only closed terms. On the other hand, in Beluga, every individual LF type is explicitly contextual—the inclusion of LF types into the computational language is the contextual modality—and there is no notion of a computational-language type in an LF context. Consequently, one must write types with the LF contexts distributed to the leaves, i.e., $(\Psi \Rightarrow A) \oplus (\Psi \Rightarrow B)$ instead of $\Psi \Rightarrow (A \oplus B)$. Types of the latter form can be more syntactically convenient (as in Delphin, a single context scopes over many LF types), but this is a minor difference, as these two types are isomorphic in our language [34]. A more significant difference is that our contextual modality $\langle \Psi \rangle\ A$ is different than that of contextual modal type theory [40] and Beluga, where contextual variables are eliminated by substitution. In previous work, we studied a framework [34] that allows inference rules with side conditions, expressed as computational functions, which place restrictions on the ambient contexts in which the rule may be applied. The presence of such side conditions 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, defined in the meta-function language. Consequently, the type theory we have presented in this paper is compatible with a future extension to a framework integrating binding and computation.

There are many techniques for representing variable binding besides LF, ranging from concrete representation techniques [5] to other theories of binding, such as nominal logic [8, 47, 56]; our previous work includes a detailed comparison with these approaches [34]. There have also been semantic studies of variable binding, both for the nominal approach [21] and for the pronominal approach where variables are projections from a context (see Fiore et al. [18] and Hofmann [28]).

Polarized intuitionistic logic has the same basic type structure ($\rightarrow$,&,$\top$,$\uparrow$,$\otimes$,1,$\oplus$,0,$\downarrow$) as call-by-push-value [32], but the programs of our calculus are different than those of CBPV, which are not fully focalized. Though our type theory exhibits the usual asymmetries of intuitionistic logic, the treatment of positive types via constructors and negative types via destructors was inspired by the duality between proofs and refutations in computational interpretations of classical logic (see, for example, Curien and Herbelin [14], Filinski [17], Selinger [51], Zeilberger [62]).

5. Conclusion

In this paper, we have generalized our previous work on computing with binding to a simple form of dependent data, yielding a language for programming with domain-specific logics. However, there are still many interesting avenues for future work:

Computation in Representation We have chosen to take LF "off the shelf" in this paper, and thus our type theory does not account for embedding computation in data (e.g., a datatype with a computational function as a component). We plan to lift this restriction in two stages: First, we can consider allowing *run-time* datatypes that mix binding and computation, while still restricting dependency to purely positive types (those with no shifts), which restricts dependency to LF-like data. More ambitiously, we may consider full-spectrum dependency on negative types as well. Full-spectrum dependency is more difficult because it imposes constraints on run-time features such as effects, but it would allow a fully integrated treatment of dependent binding and computation.

Meta-functions In this paper, we have demonstrated that our language has a suitable type structure for programming with domain-specific logics, but the examples are necessarily abstract, as we have not formally defined a finitary syntax for meta-functions τ, ψ, ϕ. We are now free to consider different implementations of meta-functions without disturbing the meta-theoretic properties of our language. For example, a simple language of meta-functions could consist of two ingredients: First, we would define 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 [16] describe some recent work addressing this problem). Second, we would 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.

Effects, Polymorphism, and Modules In scaling the calculus presented here to a full-scale programming language, we intend to investigate whether polarity and focusing offer any new insights on features such as effects, polymorphism, and modularity. In particular, we plan to consider indexing the shift types $\uparrow A^+$ and $\downarrow A^-$ to more precisely track what effects are permitted—e.g., distinguishing $\downarrow^{\mathsf{impure}} A^-$, at which general recursion is allowed, from $\downarrow^{\mathsf{pure}} A^-$, which classifies total programs. We may be able to permit programmers to reason about effectful code using domain-specific logics (e.g., coding up Ynot's separation logic [39] as a library) by indexing shifts with propositions defined in a DSL.

Acknowledgements

We thank Noam Zeilberger, Karl Crary, and Rob Simmons for discussions about this work.

References

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

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

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

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

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

[6] 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.

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

[8] J. Cheney. Simple nominal type theory. In *International Workshop on Logical Frameworks and Meta-Languages: Theory and Practice*, 2008.

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

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

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

[12] 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.

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

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

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

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

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

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

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

[20] 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.

[21] 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.

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

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

[24] 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.

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

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

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

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

[29] 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.

[30] 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.

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

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

[33] 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.

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

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

[36] 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.

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

[38] 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.

[39] 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.

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

[41] 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.

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

[43] N. Oury and W. Swierstra. The power of pi. In *ACM SIGPLAN International Conference on Functional Programming*, 2008.

[44] 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.

[45] 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.

[46] 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.

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

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

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

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

[51] 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.

[52] 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.

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

[54] 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.

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

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

[57] 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.

[58] 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.

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

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

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

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

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

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