Positively Dependent Types

Daniel R. Licata* Robert Harper *

Carnegie Mellon University

{drl,rwh}@cs.cmu.edu

Abstract

This paper is part of a line of work on using the logical techniques of polarity and focusing to design a dependent programming language, with particular emphasis on programming with deductive systems such as programming languages and proof theories. Polarity emphasizes the distinction between positive types, which classify data, and negative types, which classify computation. In previous work, we showed how to use Zeilberger's higher-order formulation of focusing to integrate a positive function space for representing variable binding, an essential tool for specifying logical systems, with a standard negative computational function space. However, our previous work considers only a simply-typed language. The central technical contribution of the present paper is to extend higher-order focusing with a form of dependency that we call *positively dependent types*: We allow dependency on positive data, but not negative computation. Additionally, we present the syntax of dependent pair and function types using an iterated inductive definition, mapping positive data to types, which gives an account of type-level computation. We construct our language inside the dependently typed programming language Agda 2, making essential use of coinductive types and induction-recursion.

Categories and Subject Descriptors F.3.3 [*Logics and Meanings Of Programs*]: Studies of Program Constructs—Type structure; F.3.1 [*Logics and Meanings Of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Verification

1. Introduction

The type systems of languages such as ML and Haskell have proved to be effective and scalable formal methods. However, some of this success has been the result of concentrating on relatively simple properties that admit automated verification with little programmer input. *Dependent types* permit the specification and verification of more-interesting program properties. The key idea of dependently typed programming is to allow indexed families of types whose indices vary with the type's inhabitants. For example, a simple list module in ML might have the following signature:

```
nil    : list
cons   : elt -> list -> list
append : list -> list -> list
nth    : nat -> list -> elt option
```

Using dependency, we can refine the the type `list` to a type family `list[n:nat]` whose index `n` represents the length of the list:

```
nil    : list[0]
cons   : Πn:nat.elt -> list[n] -> list[n+1]
append : Πn,m:nat.list[n] -> list[m] -> list[n+m]
nth    : Πn,m:nat.n<m -> list[m] -> elt
```

The types of the constructors `nil` and `cons` describe how they act on the list's length. List operations such as `append` and `nth` are given more precise types, providing more static checks on their implementations and uses. For example, `nth` is given a precondition ensuring that `n` is in bounds, and as a result no longer needs to return an `option`.

To support code like this, it is clear that a dependently typed programming language must provide (1) a class of data that can be used to index types—above, we used the natural numbers; (2) computations with such data, such as the function `+`; and (3) the ability to form indexed families of types, such as `list[n]` and `n<m`. However, there are tensions on the class of index data and computation that one provides: On the one hand, it is beneficial to provide a rich language of index data and computation to make it easy for programmers to specify and verify code. On the other, type checking depends on type equality, and type equality depends on the equality of the indices to families of types. Thus, the language of type indices is constrained by the need to compare them for equality; for example, it is difficult to allow programs that use effects such as non-termination, state, exceptions, IO, etc. at the index level, because it is difficult to compare programs using such effects for equality.

This paper is part of a line of work on using the logical techniques of polarity [20] and focusing [2] to design a dependent programming language, with particular emphasis on programming with deductive systems such as programming languages and proof theories. Our long-term goal is to allow programmers to use the tools provided by a logical framework [23] to define logics for reasoning about code, and to permit compile-time and run-time computation with such logics. Polarity emphasizes the distinction between positive types, which classify data (such as natural numbers, products and sums and lists of positive data), and negative types, which classify computation (such as functions and coinductive types). In previous work [29], we showed how to use polarity to integrate a positive function space for representing variable binding, an essential tool for specifying logical systems, with a standard negative computational function space.

* This research was sponsored in part by the National Science Foundation under grant number CCF-0702381 and by the Carnegie Mellon Anonymous Fellowship in Computer Science. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

PLPV'09, January 20, 2009, Savannah, Georgia, USA.

Our previous work follows Zeilberger's higher-order formulation of focusing [51, 52]. In this formalism, 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 from negative computation. 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, and the typing rules are defined by iterated inductive definitions [31].

However, our previous work considers only a simply-typed language. The central technical contribution of the present paper is to extend higher-order focusing with a form of dependency that we call *positively dependent types*:

1. We allow dependency only on positive data, not on negative computation.
2. The syntax of Π and Σ types is specified using an iterated inductive definition, mapping positive data to types. This allows types and indices to be computed by structural recursion.
3. Indexed families of types can be defined as indexed inductive definitions by giving pattern rules, or by recursion on indices.

Positively dependent types are sufficient for many examples, such as length-indexed lists and arrays, red-black trees that enforce the red-black invariant [14], and numbers indexed by scientific units [28]. While we do not consider our previous work's types for representing variable binding in this paper, integrating the positive representational function space with the present account of positive dependency would allow dependency on data with variable binding, as in LF [23]. The first author's thesis proposal describes applications of this kind of dependency to creating domain-specific logics for reasoning about code [27].

Positively dependent types make a syntactic phase distinction between compile-time and run-time computation: equality of types never depends on the equality of run-time computation [22]. Consequently, run-time computations in our language may make unrestricted use of effects, as in ML. This is a difference from *phase-insensitive* dependent programming languages, such as Agda [36] and Coq [5], which must treat effects much more carefully because they permit dependency on all run-time programs. We leave to future work the question of whether one can and should account for phase-insensitive dependency within our polarized formalism. On the other hand, positively dependent types allow computation with the same positive data at both compile-time and at run-time. This is a difference with many *phase-sensitive* dependent programming languages, such as DML [49] and Ωmega [44], which take the stance that compile-time data should be computationally irrelevant at run-time (though some phase-sensitive languages do allow computation with static data at both levels [13, 21, 28]). The above function `nth`, which is implemented by recursion on the same number `n` that appears in the type `n<m`, provides an example of why this is useful.

Higher-order focusing relies on an abstract notion of meta-function, which is an interface that can be realized in various ways. In this paper, we construct the language inside of Agda: our work is not just a proposal for a dependently typed programming language, but also an application of dependent programming. (The code is available from `http://www.cs.cmu.edu/~drl/`.) Our language definition makes essential use of inductive-recursive definitions [16] and coinductive types, and demonstrates that the benefits of Zeilberger's higher-order representation technique carry over to the dependently typed setting. For example, in a previous technical report [28], we presented a phase-sensitive language that supports the same caliber of dependent programming as the language described here; it required 13,000 lines of Twelf code to verify the language's type safety. The present higher-order representation, which reuses Agda functions for pattern-matching and recursion, requires less than 1000 lines of Agda to verify. As we discuss briefly below, it may also be possible to implement our design in GHC, which already implements much of the technology required to support positively dependent types.

In the remainder of this paper, we first review simply-typed polarized type theory and its Agda implementation (Section 2). Next, we describe the extension to positively dependent types (Section 3), its metatheory (Section 4), and some simple examples (Section 5).

2. Polarized Type Theory

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, 19, 24, 26, 51] 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

2.1 Higher-order Focusing

In higher-order focusing [29, 51, 52], types are specified by patterns, which are used in both focus and inversion: focus phases choose a pattern, whereas inversion phases pattern-match. Simply-typed polarized type theory is defined in three stages:

- First, we define the syntax of types.
- Next, we define patterns—constructor patterns for positive types, and destructor patterns for negative types.
- Finally, we define the focusing judgements.

It is important that patterns are defined prior to the focusing judgements, which use an iterated inductive definition quantifying over them to specify inversion.

In the remainder of this section, we review simply-typed polarized intuitionistic logic, following our previous work [29].

2.1.1 Types

The basic types of polarized type theory are the following:

$$\begin{array}{llll} \text{Type}^+ & A^+ & ::= & X^+ \mid \downarrow A^- \mid 1 \mid A^+ \otimes B^+ \mid 0 \mid A^+ \oplus B^+ \\ \text{Type}^- & A^- & ::= & X^- \mid \uparrow A^+ \mid A^+ \to B^- \mid \top \mid A^- \& B^- \end{array}$$

These types represent eager products and sums $(1, \otimes, 0, \oplus)$, lazy products $(\top, \&)$, and functions $(\to)$. The shift $\uparrow$ includes positive types into negative types, where $\uparrow A^+$ is a suspension of an expression computing an A^+. The shift $\downarrow$ includes negative values into positive values. Finally, we have atomic propositions X^+ and

Hypothesis $\alpha ::= X^+ \mid C^-$
Conclusion $\gamma ::= X^- \mid C^+$
Context $\Delta ::= \cdot \mid \Delta, x : \alpha$

$\boxed{\Delta \Vdash c :: A^+}$

$$\frac{}{x : X^+ \Vdash x :: X^+} \qquad \frac{}{x : A^- \Vdash x :: \downarrow A^-}$$

$$\frac{}{\cdot \Vdash () :: 1} \qquad \frac{\Delta_1 \Vdash c_1 :: A^+ \quad \Delta_2 \Vdash c_2 :: B^+}{\Delta_1, \Delta_2 \Vdash (c_1, c_2) :: A^+ \otimes B^+}$$

(no rule for 0) $\qquad \dfrac{\Delta \Vdash c :: A^+}{\Delta \Vdash \mathsf{inl}\ c :: A^+ \oplus B^+} \qquad \dfrac{\Delta \Vdash c :: B^+}{\Delta \Vdash \mathsf{inr}\ c :: A^+ \oplus B^+}$

$\boxed{\Delta \Vdash d :: A^- > \gamma}$

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

$$\frac{\Delta_1 \Vdash c :: A^+ \quad \Delta_2 \Vdash d :: B^- > \gamma}{\Delta_1, \Delta_2 \Vdash c\,; d :: A^+ \to B^- > \gamma}$$

(no rule for $\top$)

$$\frac{\Delta \Vdash d :: A^- > \gamma}{\Delta \Vdash \mathsf{fst};\ d :: A^- \& B^- > \gamma} \qquad \frac{\Delta \Vdash d :: B^- > \gamma}{\Delta \Vdash \mathsf{snd};\ d :: A^- \& B^- > \gamma}$$

Figure 1. Pattern rules

X^- of each polarity. Below, we will extend this language with dependency on *purely positive types*, which are those positive types that do not mention atoms X^+ or shifts $\downarrow$.

2.1.2 Patterns

Next, we define the constructor and destructor patterns in Figure 1. These judgements use the following auxiliary notions:
A hypothesis is a positive atom or a negative type; dually, a conclusion is a negative atom or a positive type. Hypotheses occur at the leaves of constructor patterns, which are defined by the judgement $\Delta \Vdash c :: A^+$. Destructor patterns have hypotheses at the leaves and additionally produce a conclusion; they are defined by the judgement $\Delta \Vdash d :: A^- > \gamma$.

The constructor patterns $\Delta \Vdash c :: A^+$ should be read as follows: the pattern for $\otimes$ is a pair of patterns; there are two patterns for $\oplus$, inl and inr. The patterns for atoms and shifts are variables, because no further pattern-matching is possible. Variables are treated linearly, because a pattern can bind a variable only once.

The destructor patterns describe the shapes of observations on a negative type A^-: $\&$ is observed by observing either the first component or the second component; suspensions $\uparrow A^+$ can be forced; functions are observed by applying them to an argument (represented by a constructor pattern) and observing the result.

2.1.3 Focusing Judgements

Next, we define the focusing judgements, presented in Figure 2. 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).

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 $\Gamma \vdash k^+ : \gamma_0 > \gamma$ defines positive continuations (left inversion): The only positive continuation for atoms X^- is the identity. A positive continuation for C^+ is a

Context^2 $\Gamma ::= \cdot \mid \Gamma, \Delta$

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

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

$\boxed{\text{Left Inversion: } \Gamma \vdash k^+ : \gamma_0 > \gamma}$

$$\frac{}{\Gamma \vdash \epsilon : X^- > X^-} \qquad \frac{(\Delta \Vdash c :: C^+ \ \longrightarrow\ \Gamma, \Delta \vdash \phi^+(c) : \gamma)}{\Gamma \vdash \mathsf{cont}^+(\phi^+) : C^+ > \gamma}$$

$\boxed{\text{Left Focus: } \Gamma \vdash k^- :: C^- > \gamma}$

$$\frac{\Delta \Vdash d :: C^- > \gamma_0 \quad \Gamma \vdash \sigma : \Delta \quad \Gamma \vdash k^+ : \gamma_0 > \gamma}{\Gamma \vdash d[\sigma]; k^+ :: C^- > \gamma}$$

$\boxed{\text{Right Inversion: } \Gamma \vdash v^- : \alpha}$

$$\frac{x : X^+ \in \Gamma}{\Gamma \vdash x : X^+} \qquad \frac{(\Delta \Vdash d :: C^- > \gamma \ \longrightarrow\ \Gamma, \Delta \vdash \phi^-(d) : \gamma)}{\Gamma \vdash \mathsf{val}^-(\phi^-) : C^-}$$

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

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

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

$$\frac{(x : \alpha \in \Delta \ \longrightarrow\ \Gamma \vdash \sigma(x) : \alpha)}{\Gamma \vdash \sigma : \Delta}$$

Figure 2. Focusing rules

case-analysis, specified by a meta-function ϕ^+ mapping patterns to expressions. We notate a meta-function by a potentially infinite set of pairs of the form $\{c_1 \mapsto e_1 \mid \ldots\}$. The premise of the rule asserts that for all constructor patterns c for C^+, applying ϕ^+ to c yields an expression of type γ using the variables bound by the pattern. We use the notation $(\mathcal{J} \longrightarrow \mathcal{J}')$ for a higher-order premise of an iterated inductive definition [31]; by convention, variables like Δ and c that occur first in the premise of an iterated inductive definition are tacitly universally quantified.

The next two judgements define focusing and inversion for the negative types. The judgement $\Gamma \vdash k^- :: C^- > \gamma$ 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 conclusion γ_0. The positive continuation reflects the fact that it may take further case-analysis of γ_0 to reach the desired conclusion γ. The judgement $\Gamma \vdash v^- : C^-$ defines negative values (right inversion): a negative value is specified by a meta-function ϕ^- that gives an expression responding to every possible destructor.

The judgement $\Gamma \vdash e : \gamma$ 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:

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

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

$$\frac{\Gamma, x_1 : A_1^- \vdash e_1 : B^+ \quad \Gamma, x_2 : A_2^- \vdash e_2 : B^+}{\Gamma \vdash \mathsf{val}^-((\mathsf{inl}\, x_1\, ; \epsilon) \mapsto e_1 \;\mid\; (\mathsf{inr}\, x_2\, ; \epsilon) \mapsto e_2) : (\downarrow A_1^-) \oplus (\downarrow A_2^-) \to \uparrow B^+}$$

2.2 Agda Implementation

Next, we describe an Agda implementation of the above type theory. We use an intrinsic encoding: rather than first defining raw syntax (c,d,v^+,k^+,...) and then defining typing judgements on raw syntax, we represent only the typing judgements, which correspond to the well-typed programs.

The correspondence between the above judgements and their Agda counterparts is as follows:

Informal	Agda
$\Delta \Vdash c :: A^+$	$\Delta \Vdash A^+$
$\Delta \Vdash d :: A^- > \gamma$	$\Delta \Vdash A^- > \gamma$
$\Gamma \vdash v^+ :: C^+$	$\Gamma \vdash \mathsf{RFoc}\, C^+$
$\Gamma \vdash k^+ : \gamma_0 > \gamma$	$\Gamma \vdash \mathsf{LInv}\, \gamma_0\, \gamma$
$\Gamma \vdash k^- :: C^- > \gamma$	$\Gamma \vdash \mathsf{LFoc}\, C^-\, \gamma$
$\Gamma \vdash e : \gamma$	$\Gamma \vdash \mathsf{Neu}\, \gamma$
$\Gamma \vdash \sigma : \Delta$	$\Gamma \vdash \mathsf{Asms}\, \Delta$

2.2.1 Types

The syntax of types is a straightforward inductive definition:

```
Atom : Set
Atom = String

mutual
 data Type⁺ : Set where
   X⁺  : Atom -> Type⁺
   ↓   : Type⁻ -> Type⁺
   1⁺  : Type⁺
   _*_ : Type⁺ -> Type⁺ -> Type⁺
   0⁺  : Type⁺
   _+_ : Type⁺ -> Type⁺ -> Type⁺

 data Type⁻ : Set where
   X⁻   : Atom -> Type⁻
   ↑    : Type⁺ -> Type⁻
   _→_  : Type⁺ -> Type⁻ -> Type⁻
   ⊤    : Type⁻
   _&_  : Type⁻ -> Type⁻ -> Type⁻
```

Using an underscore in a name is Agda syntax for mixfix operators; e.g., `&` can be used infix.

2.2.2 Patterns

On paper, we tacitly included negative types and atoms into the sum type α; in Agda, we must give injections:

```
--  α
data Hyp : Set where
  _true⁻ : Type⁻ -> Hyp
  _atom⁺ : Atom -> Hyp
-- Δ
Ctx = List Hyp
-- γ
data Conc : Set where
  _true⁺ : Type⁺ -> Conc
  _atom⁻ : Atom -> Conc
```

Constructor patterns are represented by an indexed inductive definition, indexed by the context Δ and the type `A⁺`. We write `[]` for the empty list, `++` for append, and `[ x ]` for the singleton list.

```
data _⊩_ : Ctx -> Type⁺ -> Set where
  Cx⁺ : forall {X} -> [ X atom⁺ ] ⊩ (X⁺ X)
  Cx⁻ : forall {A⁻} -> [ A⁻ true⁻ ] ⊩ (↓ A⁻)
  C<> : [] ⊩ 1⁺
  Cpair : forall {Δ₁ Δ₂ A⁺ B⁺}
        -> Δ₁ ⊩ A⁺  ->  Δ₂ ⊩ B⁺
        -> (Δ₂ ++ Δ₁) ⊩ (A⁺ * B⁺)
  Cinl : forall {Δ A⁺ B⁺}
       -> Δ ⊩ A⁺
       -> Δ ⊩ (A⁺ + B⁺)
  Cinr : forall {Δ A⁺ B⁺}
       -> Δ ⊩ B⁺
       -> Δ ⊩ (A⁺ + B⁺)

data _⊩_>_ : Ctx -> Type⁻ -> Conc -> Set where
  De⁻  : forall {X} -> [] ⊩ (X⁻ X) > (X atom⁻)
  De⁺  : forall {A⁺} -> [] ⊩ (↑ A⁺) > (A⁺ true⁺)
  Dapp : forall {Δ1 Δ2 A⁺ B⁻ γ}
       -> Δ1 ⊩ A⁺  ->  Δ2 ⊩ B⁻ > γ
       -> (Δ2 ++ Δ1) ⊩ (A⁺ → B⁻) > γ
  Dfst : forall {Δ A⁻ B⁻ γ}
       -> Δ ⊩ A⁻ > γ
       -> Δ ⊩ (A⁻ & B⁻) > γ
  Dsnd : forall {Δ A⁻ B⁻ γ}
       -> Δ ⊩ B⁻ > γ
       -> Δ ⊩ (A⁻ & B⁻) > γ
```

The Agda syntax for dependent functions may be unfamiliar to some readers: A dependent function is written `(x : A) -> B`, but the `->` between successive arguments can be elided, writing, for example, `(x : A) (y : B) -> C`. Replacing the parentheses with curly-braces, as in `{x : A} -> B`, notates an implicit argument, whose application will be inferred by unification. The syntax `forall {x1 ... xn} -> A` allows the types of the variables to be elided; it is equivalent to writing `{x1 : _} ... {xn : _} -> A`. For example, a typical application of `Dapp` would have the form `(Dapp c d)` where `c` is a constructor pattern and `d` is a destructor pattern; the contexts and types are inferred. A function can be explicitly applied to implicit arguments by enclosing the arguments in curly-braces; e.g., `(Dapp {Δ} {Δ'} c d)`.

2.2.3 Focusing Judgements

Next, we define the focusing judgements in Figure 3. It is convenient to define a sum type `FocJudg` and then define one datatype $\Gamma \vdash$ `J`, rather than giving six separate datatypes. Additionally, we use the auxiliary types $\alpha \in \Delta$ for list membership, $\alpha \in\in \Gamma$ for membership in some list in Γ, and we write `::` for consing onto a list. The list membership types serve as de Bruijn indices: a proof that $\alpha \in\in \Gamma$ is an index into Γ.

The focusing judgements are defined by an iterated inductive definition; note the higher-order premises of `Cont⁺` and `Val⁻` and `Sub`.

2.2.4 Examples

The first of the derived rules suggested above is implemented as follows:

```
ex1 : forall {Γ A1⁺ A2⁺}
    -> ([] :: Γ) ⊢ Neu (A1⁺ true⁺)
    -> ([] :: Γ) ⊢ Neu (A2⁺ true⁺)
    -> Γ ⊢ RInv ( ((↑ A1⁺) & (↑ A2⁺)) true⁻)
ex1 {Γ} {A1⁺} {A2⁺} e1 e2 = Val⁻ ex1* where
  ex1* : {Δ : Ctx}{γ : Conc}
       -> Δ ⊩ ((↑ A1⁺) & (↑ A2⁺)) > γ
       -> (Δ :: Γ) ⊢ Neu γ
```

```
-- Γ
CtxCtx : Set
CtxCtx = List Ctx

data FocJudg : Set where
  RFoc : Type⁺ -> FocJudg
  LInv : Conc -> Conc -> FocJudg
  LFoc : Type⁻ -> Conc -> FocJudg
  RInv : Hyp -> FocJudg
  Neu : Conc -> FocJudg
  Asms : Ctx -> FocJudg

data _⊢_ :  CtxCtx -> FocJudg -> Set where
  -- positive values (v+)
  Val⁺ : forall {Γ Δ C⁺}
       -> Δ ⊩ C⁺ -> Γ ⊢ Asms Δ
       -> Γ ⊢ RFoc C⁺
  -- positive continuations (k+)
  Ke⁻     : forall {Γ X}
          -> Γ ⊢ LInv (X atom⁻) (X atom⁻)
  Cont⁺  : forall {Γ γ C⁺}
          -> ({Δ : Ctx} -> Δ ⊩ C⁺ -> Δ :: Γ ⊢ Neu γ)
          -> Γ ⊢ LInv (C⁺ true⁺) γ
  -- negative continuations (k-)
  Cont⁻ : forall {Δ A⁻ γ₀ γ Γ}
        -> Δ ⊩ A⁻ > γ₀ -> Γ ⊢ Asms Δ -> Γ ⊢ LInv γ₀ γ
        -> Γ ⊢ LFoc A⁻ γ
  -- negative values (v-)
  Vx⁺ : forall {Γ X}
      -> (X atom⁺) ∈∈ Γ -> Γ ⊢ RInv (X atom⁺)
  Val⁻ : forall {Γ C⁻}
       -> ({Δ : Ctx}{γ : Conc}
           -> Δ ⊩ C⁻ > γ ->  Δ :: Γ ⊢ Neu γ)
       -> Γ ⊢ RInv (C⁻ true⁻)
  -- expressions (e)
  R : forall {Γ C⁺}
    -> Γ ⊢ RFoc C⁺
    -> Γ ⊢ Neu (C⁺ true⁺)
  L : forall {Γ C⁻ γ}
    -> ((C⁻ true⁻) ∈∈ Γ)   ->  Γ ⊢ LFoc C⁻ γ
    -> Γ ⊢ Neu γ
  -- substitutions (σ)
  Sub  : forall {Γ Δ}
       -> ({α : Hyp} -> α ∈ Δ -> Γ ⊢ RInv α)
       -> Γ ⊢ Asms Δ
```

Figure 3. Focusing judgements for simple types

```
  ex1* (Dfst De⁺) = e1
  ex1* (Dsnd De⁺) = e2
```

A lazy pair is implemented by pattern-matching against the projections and satisfying the two possible observations. This is implemented by the Agda function `ex1*`, which pattern-matches on the destructor pattern datatype. A few details deserve comment: First, we require the given expressions `e1` and `e2` to be in context `[] :: Γ` to avoid the need to call a weakening lemma. Second, the curly-braces surrounding the three arguments to `ex1` are Agda syntax for matching on implicit arguments.

The second derived rule is implemented as follows:

```
ex2 : forall {Γ A1⁻ A2⁻ B⁺}
    -> ([ A1⁻ true⁻ ] :: Γ) ⊢ Neu (B⁺ true⁺)
    -> ([ A2⁻ true⁻ ] :: Γ) ⊢ Neu (B⁺ true⁺)
    -> Γ ⊢ RInv (( ((↓ A1⁻) + (↓ A2⁻)) → (↑ B⁺))  true⁻)
ex2 {Γ} {A1⁻} {A2⁻} {B⁺} e1 e2 = Val⁻ ex2* where
  ex2* : {Δ : Ctx}{γ : Conc}
       -> Δ ⊩ (((↓ A1⁻) + (↓ A2⁻)) → (↑ B⁺)) > γ
       -> (Δ :: Γ) ⊢ Neu γ
  ex2* (Dapp (Cinl Cx⁻) De⁺) = e1
  ex2* (Dapp (Cinr Cx⁻) De⁺) = e2
```

A function of type `(((↓ A1⁻) + (↓ A2⁻)) → (↑ B⁺))` is implemented by pattern-matching against its destructor patterns, of which there is one, `Dapp`, and then further pattern-matching against the constructor patterns for the argument; there are two in this example.

3. Positively Dependent Types

The two key ideas of positively dependent types are:

1. We allow dependency only on the values of purely positive types—i.e., the shift- and atom-free positive types. Because shifts and atoms are the only types whose patterns bind variables, the values of purely positive types are closed patterns.
2. The syntax of Π and Σ types is specified using an iterated inductive definition, mapping patterns to types. Type-level pattern-matching permits types to be defined by recursion on data.

3.1 A First Attempt

We add rules for three new types: Σ^+ types; a type `nat` representing natural numbers; and a type `vec A⁺ n`, representing vectors of length `n` with elements of type `A⁺`. Here `n` is a closed `nat` pattern; i.e., it has type `[] ⊩ nat`.

Because types are dependent on patterns, and patterns are classified by types, we at least need to define the syntax of types and the pattern judgement as a mutual inductive definition.

The new types are specified as follows:

```
mutual
 data Type⁺ : Set where
   --- above constructors plus ...
   Σ⁺  : (A⁺ : Type⁺) -> ([] ⊩ A⁺ -> Type⁺) -> Type⁺
   nat : Type⁺
   vec : Type⁺ -> [] ⊩ nat -> Type⁺
```

Note the higher-order premise of Σ^+.

Next, we add pattern constructors for these types:

```
 data _⊩_ : List Hyp -> Type⁺ -> Set where
   --- above constructors plus ...
   Cdpair : forall {Δ A⁺} {τ⁺ : ([] ⊩ A⁺ -> Type⁺)}
          (c : [] ⊩ A⁺) ->  Δ ⊩ (τ⁺ c)
          -> Δ ⊩ (Σ⁺ A⁺ τ⁺)
   Czero : [] ⊩ nat
   Csucc : {Δ : Ctx}
         -> Δ ⊩ nat
         -> Δ ⊩ nat
   Cnil  : forall {A⁺}
         -> [] ⊩ vec A⁺ Czero
   Ccons : forall {A⁺ Δ₁ Δ₂}
         -> (n : [] ⊩ nat) -> Δ₁ ⊩ A⁺ -> Δ₂ ⊩ vec A⁺ n
         -> Δ₂ ++ Δ₁ ⊩ vec A⁺ (Csucc n)
```

A pattern for `(Σ⁺ A⁺ τ⁺)` consists of a closed pattern `c` for `A⁺`, as well as a pattern for `(τ⁺ c)`—i.e., the type determined for `c` by the meta-function τ^+. Meta-function application plays the role that substitution takes in the standard intro rule for Σ-types.

To illustrate type-level computation, we can define a type Σ^+ `n:nat.iszero n` that is inhabited only when the first component of the pair is `0`.

```
zpair :  Type⁺
zpair = Σ⁺ nat iszero*
  where iszero* : [] ⊩ nat -> Type⁺
        iszero* Czero     = 1⁺
        iszero* (Csucc n) = 0⁺
```

The body of the Σ^+-type is defined by pattern-matching on `nat`, mapping 0 to the unit type, and successor of anything to the void

type. Thus, it will only be possible to give a value for the second component when the first component is `Czero`.

While this definition captures the kind of dependency we want, it is not accepted by Agda because it is not strictly positive:

1. The types $\texttt{Type}^+$ and $\Delta \Vdash \texttt{A}^+$ are defined mutually inductively.
2. The premise of the constructor Σ^+ quantifies over patterns, which a negative occurrence.

Thus, it is not clear whether this definition is sensible.

3.2 Induction-Recursion

It is possible to make sense of the above definition. They key observation is that, in the syntax $\Sigma^+ A^+ \tau^+$, the meta-function τ^+ only quantifies over the patterns for a *smaller* type A^+. So the definition is staged as follows:

1. First, the type A^+.
2. Then, the patterns for A^+.
3. Then, the type $\Sigma^+ A^+ \tau^+$, which quantifies over the patterns for A^+.
4. Then, the patterns for $\Sigma^+ A^+ \tau^+$.
5. ...

This "weave" between types and their inhabitants is common in the semantics of dependent type theories (see, e.g., Constable et al. [12]); here, we use it to construct the *syntax*.

This can be formalized in Agda using a simultaneous inductive-recursive definition: we define the syntax of $\texttt{Type}^+$ inductively, while simultaneously defining a function mapping a $\texttt{Type}^+$ to the Agda `Set` classifying its patterns.

3.2.1 Positive Pattern Data

It will be useful to use a simple kinding discipline to distinguish the purely positive types, which can never have negative subcomponents embedded in them, from the positive types, which can. We write `PPos` for purely positive and `Pos` for positive, and we allow dependency only on the patterns of purely positive types.

```
data PKind : Set where
  PPos : PKind
  Pos  : PKind
```

Now, we define the Agda `Set`s used to form patterns, with one datatype for each type. The constructors are essentially those for the datatype $\Delta \Vdash \texttt{A}^+$ defined before, but abstracted over the types of the subpatterns. Because we have not yet defined the syntax of types, we use a module parametrized by the Agda `Set`'s representing them.

```
module CPats (Type⁺ : PKind -> Set) (Type⁻ : Set) where

  data Hyp : Set where
    _true⁻ : Type⁻ -> Hyp
    _atom⁺ : Atom -> Hyp

  Ctx = List Hyp

  data CPatX⁺ : Atom -> Ctx -> Set where
    Cx⁺ : forall {X} -> CPatX⁺ X [ X atom⁺ ]
  data CPat↓ : Type⁻ -> Ctx -> Set where
    Cx⁻ : forall {A⁻} -> CPat↓ A⁻ [ A⁻ true⁻ ]
  data CPat* (_⊢A⁺ : Ctx -> Set) (_⊢B⁺ : Ctx -> Set)
     : Ctx -> Set where
    Cpair : forall {Δ₁ Δ₂}
            -> Δ₁ ⊢A⁺ -> Δ₂ ⊢B⁺
            -> CPat* _⊢A⁺ _⊢B⁺ (Δ₂ ++ Δ₁)
  data CPatΣ⁺(_⊢A⁺ : Ctx -> Set)
             (_⊢τ⁺_ : Ctx -> [] ⊢A⁺ -> Set)
     : Ctx -> Set where
    Cdpair : forall {Δ}
             -> (c : [] ⊢A⁺)  ->  Δ ⊢τ⁺ c
             -> CPatΣ⁺ _⊢A⁺ _⊢τ⁺_ Δ
  data CPat1⁺ : Ctx -> Set where
    C<> : CPat1⁺ []
  data CPat0⁺ : Ctx -> Set where
  data CPat+ (_⊢A⁺ : Ctx -> Set) (_⊢B⁺ : Ctx -> Set)
     : Ctx -> Set where
    Cinl : forall {Δ} -> Δ ⊢A⁺ -> CPat+ _⊢A⁺ _⊢B⁺ Δ
    Cinr : forall {Δ} -> Δ ⊢B⁺ -> CPat+ _⊢A⁺ _⊢B⁺ Δ
  data CPatnat : Ctx -> Set where
    Czero : CPatnat []
    Csucc : forall {Δ} -> CPatnat Δ -> CPatnat Δ
  data CPatvec (_⊢A⁺ : Ctx -> Set)
     : Ctx -> CPatnat [] -> Set where
    Cnil  : CPatvec _⊢A⁺ [] Czero
    Ccons :  forall {Δ₁ Δ₂} {n : CPatnat []}
             -> Δ₁ ⊢A⁺  ->  CPatvec _⊢A⁺ Δ₂ n
             -> CPatvec _⊢A⁺ (Δ₂ ++ Δ₁) (Csucc n)
  data CPatdom (A⁻ : Type⁻) : Ctx -> Set where
    Cdom : CPatdom A⁻ [ A⁻ true⁻ ]

open CPats using (_true⁻ ; _atom⁺)
```

3.2.2 Types and Patterns

Next, we define the syntax of types and the positive patterns, using induction-recursion. There are a few differences from above. First, $\texttt{Type}^+$ is indexed by kinds. Second, we add two new types: Π^-, whose formation rule is parallel to that for Σ^+, and `dom`, which will represent the recursive type $\mu\ \texttt{D}.\ \texttt{D} \to \texttt{D}$. Its pattern rule is:

$$\frac{}{\texttt{dom} \to \uparrow \texttt{dom} \Vdash \texttt{dom}}$$

```
mutual
  data Type⁺ : PKind -> Set where
    X⁺  : Atom -> Type⁺ Pos
    ↓   : Type⁻ -> Type⁺ Pos
    1⁺  : {K : PKind} -> Type⁺ K
    _*_ : {K : PKind} -> Type⁺ K -> Type⁺ K -> Type⁺ K
    Σ⁺  : {K : PKind}
          (A⁺ : Type⁺ PPos) -> ([] ⊩ A⁺ -> Type⁺ K) -> Type⁺ K
    0⁺  : {K : PKind} -> Type⁺ K
    _+_ : {K : PKind} -> Type⁺ K -> Type⁺ K -> Type⁺ K
    nat : {K : PKind} -> Type⁺ K
    vec : {K : PKind} (A+ : Type⁺ K) -> CPatnat [] -> Type⁺ K
    dom : Type⁺ Pos

  data Type⁻ : Set where
    X⁻   : Atom -> Type⁻
    ↑    : Type⁺ Pos -> Type⁻
    _→_ : Type⁺ Pos -> Type⁻ -> Type⁻
    Π⁻   : (A+ : Type⁺ PPos) -> ([] ⊩ A+ -> Type⁻) -> Type⁻
    ⊤    : Type⁻
    _&_  : Type⁻ -> Type⁻ -> Type⁻
```

The `Set`s of patterns are defined by instantiating the above datatypes with the `Set`s determined by the recursive calls:

```
_⊩_ : {K : PKind} -> List Hyp -> Type⁺ K -> Set
Δ ⊩ (X⁺ X) = CPatX⁺ X Δ
Δ ⊩ (↓ A⁻) = CPat↓ A⁻ Δ
Δ ⊩ 1⁺     = CPat1⁺ Δ
Δ ⊩ (A⁺ * B⁺) =
  CPat* (\Δ₁ -> Δ₁ ⊩ A⁺) (\Δ₂ -> Δ₂ ⊩ B⁺) Δ
Δ ⊩ (Σ⁺ A⁺ τ+) =
  CPatΣ⁺ (\Δ -> Δ ⊩ A⁺) (\Δ c -> Δ ⊩ (τ+ c)) Δ
Δ ⊩ 0⁺ = CPat0⁺ Δ
Δ ⊩ (A⁺ + B⁺) =
  CPat+ (\Δ -> Δ ⊩ A⁺) (\Δ -> Δ ⊩ B⁺) Δ
```

```
Δ ⊩ nat = CPatnat Δ
Δ ⊩ (vec A⁺ n) = CPatvec (\Δ -> Δ ⊩ A⁺) Δ n
Δ ⊩ dom = CPatdom (dom → ↑ dom) Δ
```

The type `dom` is the recursive type μ`dom.`$\downarrow$`(dom` $\rightarrow$ $\uparrow$ `dom)`. The pattern for `dom` is a base case, with only one pattern; `Cdom` is essentially `(roll x)` for a variable `x`. We do *not* define the patterns for `dom` in terms of functions from the patterns for `dom` to the patterns for `dom`—the polarity shift caused by the $\downarrow$ halts the semantic interpretation of patterns.

3.2.3 Destructor Patterns

Next, we define the destructor patterns inductively, as before. The `Ddapp` rule is analogous to `Cdpair`: to use Π^- `A⁺` τ^-, give a constructor `c` for `A⁺`, and use $(\tau^-$ `c)`.

```
data Conc : Set where
  _true⁺ : {K : PKind} -> Type⁺ K  -> Conc
  _atom⁻ : Atom -> Conc

data _⊩_>_ : Ctx -> Type⁻ -> Conc -> Set where
  De⁻  : forall {X} -> [] ⊩ (X⁻ X) > (X atom⁻)
  De⁺  : {A⁺ : Type⁺ Pos} -> [] ⊩ (↑ A⁺) > (A⁺ true⁺)
  Dapp : forall {Δ₁ Δ₂ B⁻ γ} {A⁺ : Type⁺ Pos}
       -> Δ₁ ⊩ A⁺  ->  Δ₂ ⊩ B⁻ > γ
       -> (Δ₂ ++ Δ₁) ⊩ ((A⁺ → B⁻)) > γ
  Ddapp : forall {Δ γ} {A⁺ : Type⁺ PPos}
                 {τ⁻ : ([] ⊩ A⁺ -> Type⁻)}
        -> (c : [] ⊩ A⁺)  ->  Δ ⊩ (τ⁻ c) > γ
        -> Δ ⊩ (Π⁻ A⁺ τ⁻) > γ
  Dfst  : forall {Δ A⁻ B⁻ γ}
        -> Δ ⊩ A⁻ > γ -> Δ ⊩ (A⁻ & B⁻) > γ
  Dsnd  : forall {Δ A⁻ B⁻ γ}
        -> Δ ⊩ B⁻ > γ -> Δ ⊩ (A⁻ & B⁻) > γ
```

3.2.4 Focusing Rules

The focusing rules are essentially unchanged from the simply-typed case. The differences are confined to the implicit arguments to the constructors: every rule that binds a positive type `C⁺` now also binds a kind `K` classifying it. We present the code in Figure 4.

4. Properties

The focusing rules described above allow only canonical programs (long $\beta\eta$-normal forms). For example there is no way to a apply a value $v : \cdot \vdash \mathsf{RInv}\, A^-$ to a continuation $k : \cdot \vdash \mathsf{LFoc}\, A^- \gamma$; it is only possible to write the canonical result of this computation. Similarly, it is not obvious that it is possible to use an assumption of A^- as a negative value—i.e., to prove $A^- \vdash \mathsf{RInv}\, A^-$. These considerations motivate the cut and identity procedures, which define a canonization method for our language; they correspond to β-reduction and η-expansion, respectively.

4.1 Identity

There are two main identity principles, which are defined mutually. The negative identity says that an assumption of `C⁻` can be expanded into a negative value (right inversion). The positive identity says that there is an identity continuation (left inversion) from `C⁺` to `C⁺`. There are also three auxiliary principles: The first defines identity for assumptions α, by delegating either to the negative identity theorem, or to the primitive rule `Vx⁺` for atoms. The second maps this across all assumptions in a context, producing a substitution. The third expands conclusions γ, by delegating either to positive identity or to the rule `Ke⁻` for atoms.

The positive identity is defined to be a continuation that, when given a constructor pattern, forms the value composed of that pattern under the identity substitution (a recursive call). The negative identity is defined to be a value that, when given a destructor pattern, observes the designated variable with that destructor, under the identity substitution, and followed by the identity left-inversion (both recursive calls).

Here we write `s0` and `sS` as constructors for de Bruijn indices into Γ:

```
s0 : α ∈ Δ -> α ∈∈ (Δ :: Γ)
sS : α ∈ Γ -> α ∈  (Δ :: Γ)
```

The identity principles are defined as follows:

```
mutual
 Ke⁺ : forall {K Γ} {C⁺ : Type⁺ K}
     -> Γ ⊢ LInv (C⁺ true⁺) (C⁺ true⁺)
 Ke⁺ ~ Cont⁺ (\c -> R (Val⁺ c (Ids s0)))

 Vx⁻ : forall {Γ C⁻}
     -> ((C⁻ true⁻) ∈∈ Γ) -> Γ ⊢ RInv (C⁻ true⁻)
 Vx⁻ x ~ Val⁻ (\d -> L (sS x) (Cont⁻ d (Ids s0) Ke))

 Vx : forall {Γ α} -> (α ∈∈ Γ) -> Γ ⊢ RInv α
 Vx {α = x+ atom⁺} x ~ Vx⁺ x
 Vx {α = C- true⁻} x ~ Vx⁻ x
```

```
CtxCtx : Set
CtxCtx = List Ctx

data FocJudg : Set where
  RFoc : {K : PKind} -> Type⁺ K -> FocJudg
  LInv : Conc -> Conc -> FocJudg
  LFoc : Type⁻ -> Conc -> FocJudg
  RInv : Hyp -> FocJudg
  Neu : Conc -> FocJudg
  Asms : Ctx -> FocJudg

codata _⊢_ :  CtxCtx -> FocJudg -> Set where
  -- positive values (v+)
  Val⁺ : forall {Γ K Δ} {C⁺ : Type⁺ K}
       -> Δ ⊩ C⁺ -> Γ ⊢ Asms Δ
       -> Γ ⊢ RFoc C⁺
  -- positive continuations (k+)
  Ke⁻    : forall {Γ X }
         -> Γ ⊢ LInv (X atom⁻) (X atom⁻)
  Cont⁺  : forall {Γ K γ} {C⁺ : Type⁺ K}
         -> ({Δ : Ctx} -> Δ ⊩ C⁺ -> Δ :: Γ ⊢ Neu γ)
         -> Γ ⊢ LInv (C⁺ true⁺) γ
  -- negative continuations (k-)
  Cont⁻ : forall {Δ C⁻ γ0 γ Γ}
        -> Δ ⊩ C⁻ > γ0 -> Γ ⊢ Asms Δ -> Γ ⊢ LInv γ0 γ
        -> Γ ⊢ LFoc C⁻ γ
  -- negative values (v-)
  Vx⁺ : forall {Γ X}
      -> (X atom⁺) ∈∈ Γ -> Γ ⊢ RInv (X atom⁺)
  Val⁻ : forall {Γ C⁻}
       -> ({Δ : Ctx}{γ : Conc}
           -> Δ ⊩ C⁻ > γ ->  Δ :: Γ ⊢ Neu γ)
       -> Γ ⊢ RInv (C⁻ true⁻)
  -- expressions (e)
  R : forall {Γ K} {C⁺ : Type⁺ K}
    -> Γ ⊢ RFoc C⁺
    -> Γ ⊢ Neu (C⁺ true⁺)
  L : forall {Γ C⁻ γ}
    -> ((C⁻ true⁻) ∈∈ Γ) -> Γ ⊢ LFoc C⁻ γ
    -> Γ ⊢ Neu γ
  -- substitutions (σ)
  Sub  : {Γ : CtxCtx} -> {Δ : Ctx}
       -> ({α : Hyp} -> α ∈ Δ -> Γ ⊢ RInv α)
       -> Γ ⊢ Asms Δ
```

Figure 4. Focusing Rules

```
Ids  : forall {Δ Γ}
     -> ({α : Hyp} -> (α ∈ Δ) -> (α ∈∈ Γ))
     -> Γ ⊢ Asms Δ
Ids subset ~ Sub (\{α} i -> Vx (subset i))

Ke : forall {Γ γ} -> Γ ⊢ LInv γ γ
Ke {γ = x- atom⁻} ~ Ke⁻
Ke {γ = C+ true⁺} ~ Ke⁺
```

Do these functions terminate? There are several circumstances in which we can answer this question:

First, if the pattern judgements have the property that they decompose a type into syntactically smaller types (i.e., whenever $\Delta \Vdash A^+$, every negative type in Δ is a subexpression of A^+, and similarly for destructor patterns), then identity is total. This is because the proofs of identity make recursive calls only on the assumptions and conclusions coming from patterns. We proved this theorem in previous work [29].

However, not all types have this property. For example, `dom` violates it, because `dom` is decomposed into `dom` $\rightarrow \uparrow$ `dom`. This property is also violated by more innocuous types, such as streams specified as an inductive type with a suspended tail—i.e. the solution to `str` $\cong$ `elt *` $\downarrow\uparrow$`(str)`. The patterns for `str` produce assumptions of $\uparrow$`str`. And indeed, these types pose a problem for the identity theorem as described above: for example, the negative identity at `(dom` $\rightarrow \uparrow$ `dom)` makes a recursive call to the negative identity at `(dom` $\rightarrow \uparrow$ `dom)`!

One solution is to disallow these types. A better solution is to treat the focusing judgement $\Gamma \vdash$ `J` *coinductively*, following Girard [19], in which case identity is *productive*. The η-expansions for types like `dom` and streams are infinitely deep, but they always respond to a single observation in a finite amount of time. We have taken this solution in the Agda code: In Figure 4, we used an Agda `codata` declaration for the focusing judgement. Above, we used `~` instead of `=` for the equations defining identity, which is Agda syntax for a function whose termination should be checked by coinduction towards the result, rather than induction over the argument. Agda successfully checks these definitions, because all of the recursive calls occur under constructors.

4.2 Cut

We present the code for cut admissibility in Figure 5. We first require a weakening lemma, whose code we elide; the type Γ `⊆SS` Γ`'` classifies proofs that Γ is a subset of Γ'.

The most fundamental cuts, `Cut`$^+$ and `Cut`$^-$, put a value up against a continuation. A positive cut is reduced by applying the meta-function given in the continuation to the constructor pattern given in the value, and then applying the value's substitution to the result. A negative cut is reduced by applying the meta-function given in the value to the destructor pattern given in the continuation, and then (1) substituting into the result and (2) composing the resulting expression substitution with the positive continuation. The next three cut principles, `EK`$^+$ and `K`$^-$`K`$^+$ and `K`$^+$`K`$^+$, compose expressions and continuations. Finally, we have a substitution lemma; note that Γ `-` `i` removes the element of Γ given by the index, and that `List.SW.here?` tests whether the index into Γ is in the Δ being substituted for, and in the first case gives the index into Δ, and in the second case gives the index into Γ `-` `i`. We elide the straightforward cases of substitution. Both composition and substitution refer back to the principal cuts: composing a value with a positive continuation causes a positive cut, whereas substituting a negative value in for a variable causes a negative cut.

This cut admissibility procedure does not always terminate, corresponding to the fact that our language admits non-terminating run-time programs. We show an example of a looping program using `dom` below.

```
weaken : { Γ Γ' : CtxCtx } { J : FocJudg }
       -> Γ ⊆SS Γ' -> Γ ⊢ J -> Γ' ⊢ J

mutual
 Cut⁺  : forall {Γ K γ} {C⁺ : Type⁺ K}
       -> Γ ⊢ RFoc C⁺
       -> Γ ⊢ LInv (C⁺ true⁺) γ
       -> Γ ⊢ Neu γ
 Cut⁺ (Val⁺ c σ) (Cont⁺ φ⁺) ~ [ i0 ← σ ] (φ⁺ c)

 Cut⁻  : forall {Γ C⁻ γ}
       -> Γ ⊢ RInv (C⁻ true⁻) -> Γ ⊢ LFoc C⁻ γ
       -> Γ ⊢ Neu γ
 Cut⁻ (Val⁻ φ⁻) (Cont⁻ d σ k⁺) ~
   EK⁺ ([ i0 ← σ ] (φ⁻ d)) k⁺

 EK⁺   : forall {Γ γ0 γ}
       -> Γ ⊢ Neu γ0 -> Γ ⊢ LInv γ0 γ
       -> Γ ⊢ Neu γ
 EK⁺ (R v) k⁺ ~ Cut⁺ v k⁺
 EK⁺ (L x k⁻) k⁺ ~ L x (K⁻K⁺ k⁻ k⁺)

 K⁺K⁺ : forall {γ0 γ1 γ Γ}
      -> Γ ⊢ LInv γ0 γ1 -> Γ ⊢ LInv γ1 γ
      -> Γ ⊢ LInv γ0 γ
 K⁺K⁺ (Cont⁺ φ⁺) k2⁺ ~
   Cont⁺ (\c -> EK⁺ (φ⁺ c) (weaken sS k2⁺))
 K⁺K⁺ Ke⁻ k2⁺ ~ k2⁺

 K⁻K⁺ : forall {C⁻ γ0 γ Γ}
      -> Γ ⊢ LFoc C⁻ γ0  ->  Γ ⊢ LInv γ0 γ
      -> Γ ⊢ LFoc C⁻ γ
 K⁻K⁺ (Cont⁻ d σ k⁺) k2⁺ ~ (Cont⁻ d σ (K⁺K⁺ k⁺ k2⁺))

 [_←_]_ : forall {Δ Γ J} -> (i : Δ ∈ Γ)
        -> (Γ - i) ⊢ Asms Δ -> Γ ⊢ J
        -> (Γ - i) ⊢ J
 [ x0 ← σ0 ] L y k⁻ with List.SW.here? x0 y | σ0
 ... | Inl newy | Sub f ~ Cut⁻ (f newy) ([ x0 ← σ0 ] k⁻)
 ... | Inr newy | _     ~ L newy ([ x0 ← σ0 ] k⁻)
```

Figure 5. Cut admissibility procedure

Because cut is defined on well-typed syntax, it intrinsically ensures subject reduction, and the fact that cut can only fail by non-termination (i.e., that it coverage checks) is tantamount to a progress lemma. It is simple to adapt the cut admissibility procedure to an operational semantics on closed terms [27].

4.3 Type Equality

The focusing rules have the property that two flows of type information never meet, except at atomic propositions `X`$^+$ and `X`$^-$: in fully η-expanded form, all of the type equality tests are pushed down to base type. (For related phenomena, see LFR [30], 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). The cut principles described above have too little type information: the principal type of the cut must be annotated or guessed. On the other hand, the identity principles may be used in a situation where two different flows of type information meet. In such a circumstance, for example, we may desire a more general positive identity that `LInv` γ γ, if the surrounding context requires `LInv` γ γ`'`, for two different types γ and γ'. But when does the identity coercion defined above suffice to map one type to a different type?

For our language's types, this happens when there is an intensional mismatch in the Agda functions used to specify Σ^+ and Π^-.

For example, given an addition function `plus*` on two patterns of type `nat`, the types

```
Σ⁺ nat \n -> Σ⁺ nat \m -> vec nat (plus* m n)
                    and
Σ⁺ nat \n -> Σ⁺ nat \m -> vec nat (plus* n m)
```

will be intensionally different, but extensionally equal, in the sense that they have the same patterns. In this case, the same implementation of `Ke⁺` that we gave above will work as a coercion from one of these types to the other.

We codify this with a subtyping relation that is sufficient to define the identity coercion. Informally, the subtyping relation for positive types is defined as follows: `A1⁺ <: A2⁺` iff for every pattern `Δ1 ⊩ A1⁺`, the same pattern has type `Δ2 ⊩ A2⁺` where `Δ1 <: Δ2`. The definition is then extended to contexts, assumptions, conclusions, and negative types (which have a contravariant flip). What does it mean for the same pattern to have two different types? If we annotated the pattern judgement with raw proof terms (e.g., `Δ ⊩ c : A⁺`), then we would require the same proof term `c` in both cases. In our intrinsic encoding, we can define an auxiliary judgement `EqCPat c1 c2` which relates two patterns, with two different types, as long as they have the same structure. We also require a similar judgement for destructor patterns. The two important cases of subtyping are the following:

```
codata _<:⁺_ : {K : PKind} -> Type⁺ K -> Type⁺ K -> Set
 where
  Sub⁺ : forall {K} {A1+ A2+ : Type⁺ K} ->
   ({ Δ1 : Ctx} (c1 : Δ1 ⊩ A1+) ->
       Σ \ Δ2 -> Σ \(c2 : Δ2 ⊩ A2+)
         -> EqCPat{Δ1}{Δ2}{K}{K}{A1+}{A2+} c1 c2
            × Δ1 <:Δ Δ2)
    -> A1+ <:⁺ A2+
codata _<:⁻_ : Type⁻ -> Type⁻ -> Set where
  Sub⁻ : forall {A1- A2- } ->
    ({Δ2 : Ctx} {γ2 : Conc} (d2 : Δ2 ⊩ A2- > γ2)
      -> Σ \Δ1 -> Σ \γ1 -> Σ \(d1 : Δ1 ⊩ A1- > γ1) ->
        EqDPat d1 d2
        × ((Δ1 , γ1) <:Δγ (Δ2 , γ2)))
    -> A1- <:⁻ A2-
```

We elide the definitions of the auxiliary judgements `<:Δ` and `<:Δγ`, which define subtyping for contexts and pairs of contexts and conclusions, respectively.

The subtyping relation suffices to define the identity coercion; the code is essentially the same as above, aside from the need to push the subtyping proofs through.

```
Ke⁺ : forall {K Γ} {C1+ C2+ : Type⁺ K}
    -> C1+ <:⁺ C2+
    -> Γ ⊢ LInv (C1+ true⁺) (C2+ true⁺)

Vx⁻  : forall {Γ C1- C2- }
     -> C1- <:⁻ C2-
     -> ((C1- true⁻) ∈∈ Γ) -> Γ ⊢ RInv (C2- true⁻)
```

In summary, our focused formalism has squeezed the balloon so that proofs of equality are only needed in one spot: to show that an identity coercion exists between apparently different types. So far, we have manually constructed the subtyping proofs when necessary. However, we have also proved that the subtyping proofs are unique, in the sense that any two proofs of `A <: B` determine the same identity coercion (where "same" means structural equality of proof terms—a judgement similar to `EqCPat` but for the focusing judgement). This licenses the use of theorem proving to find proofs of subtyping, as the code determined by the proofs will be the same. We intend to explore ways of exploiting this in future work.

5. Examples

5.1 Derived Forms

First, we define some derived forms that will make writing the examples easier.

We define shorthands for case-analyzing an expression with a positive continuation, and for right-focusing on a value:

```
case_of_ : forall {Γ γ} {C+ : Type⁺ Pos}
         -> Γ ⊢ Neu (C+ true⁺)
         -> ({Δ : Ctx} -> Δ ⊩ C+ -> Δ :: Γ ⊢ Neu γ)
         -> Γ ⊢ Neu γ
case e of k⁺ = EK⁺ e (Cont⁺ k⁺)

rfv : forall {Γ K Δ} {C⁺ : Type⁺ K}
    -> Δ ⊩ C⁺ -> Γ ⊢ Asms Δ
    -> Γ ⊢ Neu (C⁺ true⁺)
rfv c σ = R (Val⁺ c σ)
```

We define utilities for creating common substitutions (definitions elided):

```
σe : (Γ : CtxCtx) -> Γ ⊢ Asms []

σid : forall {Γ Δ} -> (Δ :: Γ) ⊢ Asms Δ

σ1 : forall {Γ α} -> Γ ⊢ RInv α -> Γ ⊢ Asms [ α ]
```

The meta-functions necessary for introducing a negative value have type:

```
{Δ:Ctx}{γ:Conc} -> Δ ⊩ A⁻ > γ -> (Δ :: Γ) ⊢ γ
```

It is convenient to define a short-hand for matching on destructor patterns. For example, for the type `(A⁺ → B⁺ → ↑ C⁺)`, we will write an Agda function that takes two constructor patterns, one for `A⁺` and one for `B⁺`, rather than a function taking both constructor patterns packed as a destructor pattern. The Agda type of these convenient meta-functions is defined by induction on negative types:

```
IMetaFn⁻ : CtxCtx -> Type⁻ -> Set
IMetaFn⁻ Γ (↑ A⁺) = Γ ⊢ Neu (A⁺ true⁺)
IMetaFn⁻ Γ (A⁺ → B⁻) =
  {Δ : Ctx} -> Δ ⊩ A⁺ -> IMetaFn⁻ (Δ :: Γ) B⁻
IMetaFn⁻ Γ (Π⁻ A⁺ τ⁻) = (p : [] ⊩ A⁺) -> IMetaFn⁻ Γ (τ⁻ p)
IMetaFn⁻ Γ (X⁻ X) = Γ ⊢ Neu (X atom⁻)
IMetaFn⁻ Γ ⊤ = Unit
IMetaFn⁻ Γ (A⁻ & B⁻) = IMetaFn⁻ Γ A⁻ × IMetaFn⁻ Γ B⁻
```

These convenient meta-functions suffice to define a negative value:

```
ival⁻ : forall {Γ A} -> IMetaFn⁻ Γ A -> Γ ⊢ RInv (A true⁻)
```

5.2 Tail

First, we define a tail function on vectors of length at least one:

```
tailtp = Π⁻ nat (\n -> (vec nat (Csucc n)) → ↑(vec nat n))
tail : forall {Γ} -> Γ ⊢ RInv (tailtp true⁻)
tail {Γ} = ival⁻ tail* where
  tail* : IMetaFn⁻ Γ tailtp
  tail* n (Ccons x xs) = rfv xs (Ids {! !})
```

The negative value is defined by a two-argument meta-function, which takes a `nat` pattern `n` and pattern for `vec nat (Csucc n)`. In the `Ccons` case, we return the tail under the identity substitution. Agda's exhaustiveness checker verifies that the `Cnil` case is impossible for a vector this length.

To make the examples more readable, we leave a hole marked by `{! !}` or `?` for simple list subset relationships, which are easy but verbose to fill in; in this case, the obligation is to prove that every assumption in Δ_2 is in $(\Delta_2$ `++` $\Delta_1)$ `::` Γ. We would like to try deploying reflective theorem proving to discharge these obligations automatically.

5.3 Append

Next, we define an append function on vectors. We require an addition function on nat patterns:

```
plus* : {K : PKind}
      -> [] ⊩ nat{K} -> [] ⊩ nat{K} -> [] ⊩ nat{K}
plus* Czero n = n
plus* {K} (Csucc m) n = Csucc (plus*{K} m n)

appendtype = (Π⁻ nat (\n -> Π⁻ nat \m ->
  vec nat n → vec nat m → ↑ (vec nat (plus*{PPos} n m))))
append : forall {Γ} -> Γ ⊢ RInv (appendtype true⁻)
append {Γ} = ival⁻ append* where
  append* : IMetaFn⁻ Γ appendtype
  append* Czero m Cnil l2 = rfv l2 σid
  append* (Csucc n) m (Ccons x l1) l2 =
    case (weaken {! !} (append* n m l1 l2)) of
      \l12 -> rfv (Ccons x l12) (Ids {! !})
```

The meta-function `append*` is defined recursively: we are using induction in Agda to do induction in the object language. The focusing syntax makes the evaluation-order explicit in the second case: do the recursive call, and then compose the result with a continuation that conses `x` onto the result.

5.4 Map

We define a function that maps a two-argument function across two lists of the same length:

```
map2type = Π⁻ nat (\n -> (↓ (nat → nat → ↑ nat) →
  vec nat n → vec nat n → ↑ (vec nat n)))
map2 : forall {Γ} -> Γ ⊢ RInv (map2type true⁻)
map2 {Γ} = ival⁻ map2* where
  map2* : IMetaFn⁻ Γ map2type
  map2* Czero Cx⁻ Cnil Cnil = rfv Cnil (σe _)
  map2* (Csucc n) Cx⁻ (Ccons x xs) (Ccons y ys) =
    case weaken {! !} (map2* n Cx⁻ xs ys) of
     \t ->
       L (sS (sS (sS (s0 i0))))
       (Cont⁻ (Dapp x (Dapp y De⁺)) (Ids {! !})
              (Cont⁺ (\h -> rfv (Ccons h t) (Ids {! !}))))
```

In the second case, we let `t` be the result of mapping the given function across the tails of the lists, then we let `h` be the result of calling the function on the heads, and finally we cons `h` onto `t`. Agda's exhaustiveness checker verifies that the nil/cons and cons/nil cases are impossible because the lists have the same length.

5.5 Map2app

To illustrate the need for proofs of type equality, we implement a function `map2app` of type

```
map2apptp =
 Π⁻ nat (\n -> Π⁻ nat (\m ->
   ↓ (nat → nat → ↑ nat)
   → vec nat n → vec nat m
   → ↑ (vec nat (plus*{PPos} n m))))
```

Informally, this function is defined as follows:

```
map2app n m f l1 l2 =
  case (append n m l1 l2) of
    app1 => case (append m n l2 l1) of
              app2 => map2 (plus* n m) f app1 app2
```

I.e., we map the given function across the results of appending the two lists in both orders. However, the second list `app2` has type `vec nat (plus* m n)`, whereas it is expected to have type `vec nat (plus* n m)`. Consequently, it is necessary to prove a subtyping relationship inductively:

```
comm : {A⁺ : Type⁺ Pos} (n m : [] ⊩ nat{PPos}) ->
     (vec A⁺ (plus*{PPos} m n))
 <:⁺ (vec A⁺ (plus*{PPos} n m))

commext : {A⁺ : Type⁺ Pos} ->
  (Σ⁺ nat \n -> Σ⁺ nat \m -> (vec A⁺ (plus*{PPos} m n)))
 <:⁺ (Σ⁺ nat \n -> Σ⁺ nat \m -> (vec A⁺ (plus*{PPos} n m)))
```

Then we coerce `app2` by cutting with the identity coercion induced by this subtyping proof, and call the result `app2'`:

```
map2app : forall {Γ} -> Γ ⊢ RInv (map2apptp true⁻)
map2app {Γ} = ival⁻ map2app* where
  map2app* : IMetaFn⁻ Γ map2apptp
  map2app* n m Cx⁻ l1 l2 =
   Cut⁻ append
    (Cont⁻ (Ddapp n (Ddapp m (Dapp l1 (Dapp l2 De⁺))))
           (Ids {! !}) (Cont⁺ (\app1 ->
   Cut⁻ append
    (Cont⁻ (Ddapp m (Ddapp n (Dapp l2 (Dapp l1 De⁺))))
           (Ids {! !}) (Cont⁺ (\app2 ->
      case (Cut⁺ (Val⁺ app2 (Ids {! !}))
                 (Ident.Ke⁺ (comm {nat} n m))) of
        \app2' ->
         Cut⁻ map2
             (Cont⁻ (Ddapp (plus* n m)
                     (Dapp Cx⁻
                      (Dapp app1 (Dapp app2' De⁺))))
                    (Ids ?) Ke)))))))
```

5.6 Loop

To illustrate that our language is compatible with effects such as non-termination, we write a loop using `dom`. The essential idea is a variation of $(\lambda x.\,x\ x)(\lambda x.\,x\ x)$:

```
loop (Cdom f) = f (Cdom f)
explode = loop (Cdom loop)
```

This code is represented as follows:

```
loop : forall {Γ} -> Γ ⊢ RInv ((dom → ↑ dom) true⁻)
loop {Γ} = ival⁻ (\{_} -> loop*) where
  loop* : IMetaFn⁻ Γ (dom → ↑ dom)
  loop* Cdom = L (s0 i0)
        (Cont⁻ (Dapp Cdom De⁺)
               (σ1 (Vx (s0 i0))) Ke)

explode : [] ⊢ Neu (dom true⁺)
explode = Force.force
 (Cut⁻ loop (Cont⁻ (Dapp Cdom De⁺) (σ1 loop) Ke))
```

The function `Force.force` pattern-matches on the focusing term, which causes Agda to evaluate it and loop.

6. 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, 8, 9, 10, 11, 15, 17, 18, 32, 35, 36, 37, 42, 43, 44, 45, 47, 48, 50, 53], building on dependently typed proof assistants such as NuPRL [12] and Coq [5]. Our central technical contribution relative to these type theories is to show how to support a form of dependency within the formalism of higher-order focusing.

Positively dependent types are slightly more parsimonious than those phase-sensitive languages which duplicate types like the natural numbers at the static and dynamic levels [8, 9, 15, 18, 37, 42, 44, 47, 48, 50, 53]. On the other hand, the reason for this duplication is that these languages commit to the computational irrelevance of static data, which gives programmers a way, albeit relatively coarse-grained, to indicate what data should be passed at

run-time. We plan to address this issue both with compiler optimizations that require no programmer input (see Brady [6]) and by integrating proof irrelevance [4, 33, 38] into the type structure of our language, which will provide more fine-grained control. Other languages, such as Delphin [41] and Beluga [39], allow dependency on LF terms, and allow run-time computation with this data, but do not provide for compile-time computation with it.

Unlike phase-insensitive languages [3, 17, 32, 35, 36, 45], positively dependent types make a syntactic distinction between run-time and compile-time computations, and thus are compatible with standard treatments of effects, as in ML. Positively dependent types are thus a useful first step that can be taken while research on methods of encapsulating effects progresses. One relatively coarse solution is to confine all effects, including non-termination, to an `IO` monad. Many more-refined treatments are possible, based on separating different effects into different monads [40], treating some effects comonadically [34], reasoning about effects using purely functional models [46], and reasoning about effects using specification logics [35]. None of these approaches yet provide a satisfactory account of benign effects, where effects are used under-the-hood to implement a pure interface (e.g., splay trees).

Polarized intuitionistic logic has the same basic type structure as call-by-push-value [25], but the programs of our calculus are different than those of CBPV, which are not fully focalized. Polarized classical logical has been applied to programming in a variety of ways; e.g, to analyze evaluation order [51] and to emphasize connections with game-theoretic semantics [24]. However, none of these polarized type theories consider dependent types.

7. Conclusion

In this paper, we have demonstrated how to support a simple but useful form of dependency, positively dependent types, within the formalism of higher-order focusing. There are many directions for future work. One is to integrate the dependency considered here with our previous work [29], which considers a positive function space used to represent variable binding. This would allow dependency on data with variable binding, as in LF, which is useful for representing logical systems. Another direction is to extend our polarized type theory with ways of encapsulating effects. This would provide an account of total programming and proof alongside our current computational language, which allows unrestricted effects, and is necessary for considering dependency on negative computations.

Finally, we may consider an implementation of positively dependent types as an extension of ML or Haskell. GHC already implements much of the technology necessary: Associated type synonyms [7] provide a notion of type-level functions which could be used to realize type-level meta-functions, and of course Haskell supports run-time functions defined by pattern-matching, which can realize the value-level meta-functions. GHC also implements indexed datatypes (GADTs), but at present these datatypes can only be indexed by constructors of kind type. Our proposal is to allow these datatypes to additionally be indexed by the values of purely positive types. The primary difficulty is that positive types are hard to come by in Haskell: tuples are more $\&$-like than $\otimes$-like, and sums build in shifts (Either $A^- \; B^-$ can be read as $\uparrow(\downarrow A^- \oplus \downarrow B^-)$). One solution would be to allow dependency only on datatypes with strictness annotations at each recursive call (`data Nat = Z | S !Nat`) and to treat the dependent function type Π^- as call-by-value, rather than call-by-name. Adding positively dependent types to ML would be easier, as ML sums and products of purely positive types are purely positive.

Acknowledgements

We thank Noam Zeilberger for discussions about this work, and we thank the anonymous reviewers for their helpful feedback.

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] S. Awodey and A. Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

[5] 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.

[6] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[7] M. Chakravarty, G. Keller, and S. P. Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, 2005.

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

[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] K. Crary and S. Weirich. Flexible type analysis. In *International Conference on Functional Programming*, 1999.

[14] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, August 2007. CMU-CS-07-129.

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

[16] P. Dybjer and A. Setzer. Indexed induction-recursion. In *Proof Theory in Computer Science*, pages 93–113. Springer, 2001.

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

[18] 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.

[19] 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.

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

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

[22] R. Harper, J. C. Mitchell, and E. Moggi. Higher-order modules and the phase distinction. In *Symposium on Principles of Programming Languages*, 1990.

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

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

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

[26] 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.

[27] D. R. Licata. Dependently typed programming with domain-specific logics (thesis proposal). Available from `http://www.cs.cmu.edu/~drl`, 2008.

[28] D. R. Licata and R. Harper. A formulation of Dependent ML with explicit equality proofs. Technical Report CMU-CS-05-178, Department of Computer Science, Carnegie Mellon University, 2005.

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

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

[31] 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.

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

[33] A. Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *International Conference on Typed Lambda Calculi and Applications*, 2001.

[34] A. Nanevski. A modal calculus for exception handling. In *Intuitionistic Modal Logic and Applications*, 2005.

[35] 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.

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

[37] 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.

[38] F. Pfenning. Intensionality, extensionality, and proof irrelevance in modal type theory. In *IEEE Symposium on Logic in Computer Science*, 2001.

[39] 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.

[40] G. Plotkin and J. Power. Notions of computation determine monads. In *Proc. FOSSACS 2002, Lecture Notes in Computer Science 2303*, pages 342–356. Springer, 2002.

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

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

[43] 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.

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

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

[46] W. Swierstra and T. Altenkirch. Beauty in the beast: A functional semantics of the awkward squad. In *ACM SIGPLAN Workshop on Haskell*, pages 25–36, 2007.

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

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

[49] H. Xi and F. Pfenning. Dependent types in practical programming. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

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

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

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

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