Refinement Types and Computational Duality

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

Abstract

One lesson learned painfully over the past twenty years is the perilous interaction of Curry-style typing with evaluation order and side-effects. This led eventually to the value restriction on polymorphism in ML, as well as, more recently, to similar artifacts in type systems for ML with intersection and union refinement types. For example, some of the traditional subtyping laws for unions and intersections are unsound in the presence of effects, while union-elimination requires an *evaluation context restriction* in addition to the value restriction on intersection-introduction.

Our aim is to show that rather than being ad hoc artifacts, phenomena such as the value and evaluation context restrictions arise naturally in type systems for effectful languages, out of principles of duality. Beginning with a review of recent work on the Curry-Howard interpretation of *focusing proofs* as pattern-matching programs, we explain how to interpret intersection and union refinements on these programs, and how to *logically derive* the subtyping relationship via an identity coercion interpretation. The value restriction, etc., emerge out of this analysis. However, this abstract account does not immediately yield a decidable type system, essentially because the syntax is infinitary—both "infinitely wide" and "infinitely deep". We show how to mechanically construct a finitary syntax by applying two well-known PL techniques—pattern-compilation and defunctionalization—and conclude by giving this finitary syntax an algorithmic refinement type system. Parallel to the text, we describe an embedding in the dependently-typed functional language Agda, both for the sake of clarifying these ideas, and also because formalization was an important guide in developing them. As one example, the Agda encoding split very naturally into an intrinsic ("Church") view of well-typed programs, and an extrinsic ("Curry") view of refinement typing for those programs.

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages; F.4.1 [*Mathematical Logic and Formal Languages*]: Proof theory

General Terms Languages

1. Introduction

In the 1972 paper "Definitional Interpreters for Higher-Order Programming Languages", John Reynolds wrote:

> Purely applicative languages are often said to be based on a logical system called the lambda calculus, or even to be "syntactically sugared" versions of the lambda calculus... However, as we will see, although an unsugared applicative language is syntactically equivalent to the lambda calculus, there is a subtle semantic difference. Essentially, the "real" lambda calculus implies a different "order of application" (i.e., normal-order evaluation) than most applicative programming languages.

This "subtle semantic difference" has turned out to be a real source of pain for the descendants of lambda calculus. A well-known product of this pain is the so-called *value restriction* in ML. Before the value restriction was adopted, various implementations of ML were unsound due to the interaction of polymorphism with side-effects. For example, in early compilers, the following clearly senseless program passed the typechecker, with the variable `x` given polymorphic type `('a list) ref`:

```
let val x = ref []
in (x := ["hello world!\n"]; hd (!x) + 2)
end
```

This was thought to be an issue peculiar to mutable references, and various patches to typechecking were proposed. However the problem was more pervasive, as Harper and Lillibridge (1991) exhibited in 1991 with another unsound but (at the time) well-typed program, using SML/NJ's `callcc` feature. The value restriction, proposed by Wright (1995), ruled out such programs by limiting polymorphic generalization to syntactic values (e.g., the expression `ref []` above is not a value, because at run-time it evaluates to a fresh location storing the empty list).

Nor is the issue peculiar to polymorphism: in general, it arises in type systems that begin to capture more precise semantic properties of programs, and in particular in *refinement type systems* (Freeman and Pfenning 1991). For example, using counterexamples similar to the above, Davies and Pfenning (2000) noticed that a value restriction is also needed for intersection types in effectful call-by-value languages, and moreover that some standard distributivity laws of subtyping (Barendregt et al. 1983) [$(A \to B) \cap (A \to C) \leq A \to (B \cap C)$ and $\top \leq A \to \top$] are unsound in that setting. More surprisingly, it turns out that a dual, *evaluation context restriction* is also necessary for eliminating (untagged) union types. In an effect-free setting, Barbanera et al. (1995) had studied union types with the following elimination rule:

$$\frac{\Gamma \vdash e : A \cup B \quad \Gamma, x : A \vdash e' : C \quad \Gamma, x : B \vdash e' : C}{\Gamma \vdash e'[e/x] : C}$$

The rule allows eliminating arbitrarily many occurrences of an expression e of union type, by discriminating the union. But Dunfield and Pfenning (2004) found this was unsound in the presence of effects, since the different occurrences of e could evaluate to a value of type A or B nondeterministically. Therefore they proposed the unusual "tridirectional" rule, schematized by evaluation contexts $E[]$:

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

$$\frac{\Gamma \vdash e : A \cup B \quad \Gamma, x : A \vdash E[x] : C \quad \Gamma, x : B \vdash E[x] : C}{\Gamma \vdash E[e] : C}$$

This rule only allows eliminating a single occurrence of e, in evaluation position. Although Dunfield and Pfenning did not mention it explicitly, the reasons for the evaluation context restriction also imply that standard laws [$(A \to C) \cap (B \to C) \leq (A \cup B) \to C$ and $\top \leq \bot \to C$] are unsound for call-by-*name* functions in the presence of effects (the latter even when the only effect is nontermination).

All of these restrictions were discovered by "disillusionment", so to speak, in the sense that the messy world of side-effects provided counterexamples to simple but naive rules. Yet, bitter experience is only a poor substitute for theoretical foundations, and understandably we may get the feeling that policies such as the value and evaluation context restrictions amount to ad hoc *monsterbarring.*[1] The central aim of this paper is to explain why phenomena such as the value and evaluation context restrictions, as well as evaluation-order-dependent subtyping laws, need not be policies imposed after-the-fact, but instead can arise synthetically from a logical view of refinement typing—our goal is not only to better understand existing choices, but to develop a theoretical framework that narrows the design space for future, more expressive type systems for effectful programming languages.

The starting point for our explanation is the rich account of logical duality provided by *focusing proofs* (Andreoli 1992; Girard 2001), and its connection to computational duality—in the sense of Filinski (1989), Curien and Herbelin (2000), Selinger (2001), and others—via the Curry-Howard correspondence. I described this connection in recent work (Zeilberger 2008a,b). The first half of this paper (§2 and §3) gives a simple recipe for intersection and union refinement types on focusing proofs-as-programs, and explains how this suffices for obtaining a rational reconstruction of all of the above phenomena. The value restriction emerges as a default, and the question becomes when a rule for typing non-values is logically derivable. Rather than introducing a separate set of rules for subtyping (and having to prove them sound), we *reduce* subtyping to typing via an *identity coercion interpretation* (§3.4). In order to identify and address the issue of *unsafe* subtyping laws, the identity coercion interpretation is also related to a *no-counterexamples interpretation* (§3.6, although our results here are only partial). Key to the overall simplicity of our explanation of refinement typing is that the Curry-Howard interpretation of focusing provides "pattern-matching for free" using a potentially infinitary, higher-order syntax. However, as we will see, this abstract account has the drawback that, in the general case, refinement checking is undecidable.

Essentially, the problem is that the syntax admits both "infinitely wide" and "infinitely deep" terms. To resolve these two issues, we borrow two well-known programming languages techniques: pattern-compilation and defunctionalization. The latter part of the paper (§4 and §5) explains how to systematically construct a finitary syntax by applying these transformations, and then how to provide *this* syntax with an algorithmic refinement checking system. Undecidability is avoided by placing annotations on *cuts.*

Both the infinitary and finitary systems have been formally represented in the dependently-typed language Agda (Norell 2007), and we describe pieces of this encoding in parallel to the text—not only for the sake of clarifying these concepts, but also for the sake of methodological transparency, since formalization was a helpful constraint in developing these ideas. As one example, the Agda encoding split very naturally into an intrinsic ("Church") view of well-typed programs, and an extrinsic ("Curry") view of refinement typing for those programs—a conceptual division recently argued for by Pfenning (2008), and earlier pondered by Reynolds (2000).

[1] Thanks to Neel Krishnaswami for pointing me to this wonderful verb in Lakatos' *Proofs and Refutations*.

2. Background: focusing proofs-as-programs

In this section we review the Curry-Howard interpretation of focusing proofs, detailed in (Zeilberger 2008a,b).

2.1 A quick summary

Underlying focusing are two related forms of logical duality: between proof and refutation, and between positive and negative *polarity.* Intuitively, if the truth or falsehood of a proposition A is thought of as the result of a game between Prover and Refuter, then the polarity of A tells us who has the first move. Formally, focusing distinguishes between two canonical modes of inference: *focus* and *inversion.* Focus corresponds to giving direct evidence for an inference (that A is true or false), while inversion corresponds to matching against all possible forms of direct evidence, and deriving a contradiction.

These "forms of direct evidence" can be axiomatized as *proof patterns* p and *refutation patterns* q. The focusing discipline can then be summarized in a square:

	Proof	Refutation
Positive	$\exists$p	$\forall$p
Negative	$\forall$q	$\exists$q

We read this as follows: 1. For positive A, a proof of A may choose any *particular* proof pattern (and fill in any holes), while a refutation of A must consider *all* possible proof patterns (and derive a contradiction from each); 2. Dually, for negative A, a refutation of A chooses a particular refutation pattern (filling in any holes), while a proof considers all possible refutation patterns (deriving contradiction from each). The power of focusing is that the rules for building proofs and refutations can be thus stated generally, with the definition of particular logical connectives entirely contained within the rules for forming proof/refutation patterns.

Computationally, a proof pattern corresponds to what we ordinarily think of as "pattern" in functional programming, i.e., a tree of constructors (with variables at the leaves). A refutation pattern corresponds to a less familiar notion: a tree of destructors. Actual proofs and refutations correspond to *values* and *continuations.* The polarity of a type tells us whether it is *strict* (positive) or *lazy* (negative), and how to analyze the values and continuations of that type: 1. A value of positive type is a pattern (of constructors) together with a *substitution* filling in the pattern's variables, while a strict continuation is a map from patterns to *expressions*; 2. A continuation of negative type is a pattern of destructors together with a substitution, while a lazy value is a map from destructor patterns to expressions. Since focusing proofs define a sequent calculus, cut-elimination unambiguously defines the operational semantics of programs. In fact, the focusing discipline naturally enforces continuation-passing-style, and we can see the polarity of a type as corresponding to the choice of call-by-value or call-by-name CPS translations.

Since most things we want to know about the negative row of the square we can learn by mechanically dualizing the corresponding statement about the positive row, we will limit attention to positive polarity in this paper. The reader can consult Zeilberger (2008a) for a more detailed explanation of negative (and mixed positive/negative) polarity.

2.2 Connectives and patterns

Definition 1 (Contexts and variables)**.** *A* **context** Δ *is a multiset of hypotheses of the form "A false", where A is positive. A particular hypothesis A false* $\in \Delta$ *is called a* **continuation variable** κ*.*

Positive connectives are *defined* by the judgment $\Delta \Vdash A\ \mathit{true}$, which asserts, intuitively, that a proof of A is directly obtainable from premises Δ, using them linearly. For example, we define negation, strict products and sums as follows:

$$\frac{}{A\ \mathit{false} \Vdash {}^{v}\!\neg A\ \mathit{true}} \qquad \frac{}{\cdot \Vdash 1\ \mathit{true}} \qquad \frac{\Delta_1 \Vdash A\ \mathit{true} \quad \Delta_2 \Vdash B\ \mathit{true}}{\Delta_1, \Delta_2 \Vdash A \otimes B\ \mathit{true}}$$

$$\text{(no rule for 0)} \qquad \frac{\Delta \Vdash A\ \mathit{true}}{\Delta \Vdash A \oplus B\ \mathit{true}} \qquad \frac{\Delta \Vdash B\ \mathit{true}}{\Delta \Vdash A \oplus B\ \mathit{true}}$$

Note that we are adopting linear logic convention in writing $\otimes$ and $\oplus$ for positive conjunction and disjunction, and are writing "v" over the negation symbol to reflect the fact that it is a call-by-$v$alue negation.

Definition 2 (Patterns)**.** *A derivation $p :: (\Delta \Vdash A\ \mathit{true})$ is called a* **pattern***, or more specifically, an A-pattern.*

Patterns correspond to patterns in the usual functional programming sense, with the proviso that they go "as deep as possible", i.e., up to continuation variables.[2] To make this more apparent, we can annotate the above rules like so:

$$\frac{}{\_ :: (A\ \mathit{false} \Vdash {}^{v}\!\neg A\ \mathit{true})}$$

$$\frac{}{() :: (\cdot \Vdash 1\ \mathit{true})} \qquad \frac{p_1 :: (\Delta_1 \Vdash A\ \mathit{true}) \quad p_2 :: (\Delta_2 \Vdash B\ \mathit{true})}{(p_1, p_2) :: (\Delta_1, \Delta_2 \Vdash A \otimes B\ \mathit{true})}$$

$$\frac{p :: (\Delta \Vdash A\ \mathit{true})}{\mathsf{inl}\, p :: (\Delta \Vdash A \oplus B\ \mathit{true})} \qquad \frac{p :: (\Delta \Vdash B\ \mathit{true})}{\mathsf{inr}\, p :: (\Delta \Vdash A \oplus B\ \mathit{true})}$$

Additional connectives can be included in a modular way by adding their pattern-formation rules, and we will do so in §2.5.

Strictly speaking, in this paper we take a de Bruijn view of variables and patterns. However, we will sometimes find it evocative to think of variables as living in/bound by patterns, replacing the placeholder _ with a concrete κ. The formal meaning of this notation should always be clear from context.

2.3 Values, continuations, substitutions, expressions

Suppose we have defined some positive connectives and their patterns. Now we can explain how to build actual proofs and refutations. We write $A\ \mathit{true}$ and $A\ \mathit{false}$ for the judgments that A is true or false, and in addition write Δ to assert the conjunction of all its basic hypotheses, and $\#$ to assert contradiction. These four judgments

$$J \quad ::= \quad A\ \mathit{true} \mid A\ \mathit{false} \mid \Delta \mid \#$$

are defined hypothetically, with respect to a **context list** Γ:

$$\Gamma \quad ::= \quad \cdot \mid \Gamma, \Delta$$

Intuitively, the hypothetical $\Gamma \vdash J$ asserts J under assumption of all hypotheses in all contexts in Γ. We write $A\ \mathit{false} \in \Gamma$ as shorthand for $A\ \mathit{false} \in \Delta \in \Gamma$.

The formal meaning of the hypothetical judgment is defined by *canonical derivations:* for each judgment J, we give a single rule describing how to derive $\Gamma \vdash J$. Before describing these, we ask the reader to absorb the following suggestive terminology:

Definition 3 (Terms)**.** *A derivation $t :: (\Gamma \vdash J)$ is called a* **term.** *A term may be further classified as follows:*

- *A* **value** *is a derivation $V :: (\Gamma \vdash A\ \mathit{true})$*
- *A* **continuation** *is a derivation $K :: (\Gamma \vdash A\ \mathit{false})$*
- *A* **substitution** *is a derivation $\sigma :: (\Gamma \vdash \Delta)$*
- *An* **expression** *is a derivation $E :: (\Gamma \vdash \#)$*

[2] Lassen and Levy (2007) independently defined an analogous notion, which they call "ultimate patterns".

$$\begin{array}{lll} \text{Contexts} & \Delta & ::= \ \cdot \mid \Delta, A\ \mathit{false} \\ & \Gamma & ::= \ \cdot \mid \Gamma, \Delta \\ \text{Judgments} & J & ::= \ A\ \mathit{true} \mid A\ \mathit{false} \mid \Delta \mid \# \end{array}$$

$$\frac{\Delta \Vdash A\ \mathit{true} \quad \Gamma \vdash \Delta}{\Gamma \vdash A\ \mathit{true}} \qquad \frac{\Delta \Vdash A\ \mathit{true} \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A\ \mathit{false}}$$

$$\frac{A\ \mathit{false} \in \Delta \quad \longrightarrow \quad \Gamma \vdash A\ \mathit{false}}{\Gamma \vdash \Delta} \qquad \frac{A\ \mathit{false} \in \Gamma \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash \#}$$

$$\frac{}{\Gamma \vdash \#}\ \mho$$

Figure 1. The rules of focusing, plus the *daimon* rule

A value (i.e., a direct proof of A) is given by a pair of an A-pattern with a substitution:

$$\frac{\Delta \Vdash A\ \mathit{true} \quad \Gamma \vdash \Delta}{\Gamma \vdash A\ \mathit{true}}$$

This rule has two premises; if we label the first by p and the second by σ, the conclusion is annotated $p[\sigma]$.

A continuation (i.e., a refutation of A) is specified by a map from any A-pattern to an expression:

$$\frac{\Delta \Vdash A\ \mathit{true} \quad \longrightarrow \quad \Gamma, \Delta \vdash \#}{\Gamma \vdash A\ \mathit{false}}$$

This rule, in the notation of Martin-Löf (1971), can have any number of premises $\Gamma, \Delta \vdash \#$, one for each derivation of $\Delta \Vdash A\ \mathit{true}$. Alternatively, we can think of it as a *single* premise demanding an *implication:* for any A-pattern $p :: (\Delta \Vdash A\ \mathit{true})$, there exists an expression $E_p :: (\Gamma, \Delta \vdash \#)$. We call such an implication, proven by arbitrary means in the metalogic, a **metafunction**. The precise definition of the metalogic is deliberately open-ended, because for the most part we will deal with metafunctions purely by their extension on patterns, only abandoning this principle when it gets in the way of decidability (see §3.8). So we simply annotate the conclusion of this rule $\mathsf{con}(\varphi)$, where φ denotes the metafunction.

A substitution (i.e., a derivation of a conjunction of hypotheses) is created by mapping every continuation variable in Δ to a continuation:

$$\frac{A\ \mathit{false} \in \Delta \quad \longrightarrow \quad \Gamma \vdash A\ \mathit{false}}{\Gamma \vdash \Delta}$$

Again, the premise of this rule demands proving an implication, that for any continuation variable $\kappa :: (A\ \mathit{false} \in \Delta)$, we can build a corresponding continuation $K_\kappa :: (\Gamma \vdash A\ \mathit{false})$. If we label this premise ρ, the conclusion is annotated $\mathsf{sub}(\rho)$.

Finally, an expression (i.e., a contradiction) is specified by selecting a continuation variable and throwing it a value:

$$\frac{A\ \mathit{false} \in \Gamma \quad \Gamma \vdash A\ \mathit{true}}{\Gamma \vdash \#}$$

If the first premise is labeled κ and the second V, the conclusion is annotated $\kappa\, V$.

The four rules for forming values, continuations, substitutions, and expressions are summarized in Figure 1.

2.4 Operational semantics: identity and cut

Although it might not be immediately obvious, the judgments above precisely define a focusing strategy, in the sense of Andreoli, for the (propositional) classical sequent calculus. To see this, just go through the following simple syntactic transformation:

- A context Γ of hypotheses becomes a simple list of formulas $|\Gamma|$, where we erase the "$-$ *false*" annotations.
- $\Gamma \vdash A$ *true* becomes a sequent under right-focus $\vdash |\Gamma|; A$
- $\Gamma \vdash A$ *false* becomes a sequent under left-inversion $A \vdash |\Gamma|$
- $\Gamma \vdash \#$ becomes the neutral sequent $\vdash |\Gamma|$
- $\Gamma \vdash \Delta$ is expanded into a list of sequents under left-inversion

Since the system defines a sequent calculus, we should expect it to satisfy the usual important conditions, such as the subformula property, expansion of identity axioms, and reduction of cuts. The subformula property is immediate given the following abstract definition of subformula:

Definition 4 (Subformula)**.** *B is called a* **subformula** *of A (written $B \prec A$) if there is some pattern $\Delta \Vdash A$ true such that B false $\in \Delta$.*

Proposition 5. *Any derivation of $\Gamma \vdash J$ mentions only subformulas of formulas in Γ and J.*

Identity and cut are stated as follows:

Principle 6 (Identity)**.** *For all Γ, Δ, A:*

1. *If A false $\in \Gamma$ then $\Gamma \vdash A$ false*
2. *$\Gamma, \Delta \vdash \Delta$*

Principle 7 (Cut)**.** *For all Γ, Δ, A, J:*

1. *If $\Gamma \vdash A$ false and $\Gamma \vdash A$ true then $\Gamma \vdash \#$*
2. *If $\Gamma \vdash \Delta$ and $\Gamma, \Delta \vdash J$ then $\Gamma \vdash J$*

It is best to see these principles annotated with terms—identity corresponds to the following mutually recursive terms Id_κ and *id*, and cut to the mutually recursive terms $K \bullet V$ and $t[\sigma]$:

- $Id_\kappa = \mathsf{con}(p \mapsto \kappa\,(p[id])) :: (\Gamma \vdash A\ \mathit{false})$, where $\kappa :: (A\ \mathit{false} \in \Gamma)$
- $id = \mathsf{sub}(\kappa \mapsto Id_\kappa) :: (\Gamma, \Delta \vdash \Delta)$
- $K \bullet V = \varphi(p)[\sigma] :: (\Gamma \vdash \#)$, where $K = \mathsf{con}(\varphi) :: (\Gamma \vdash A\ \mathit{false})$ and $V = p[\sigma] :: (\Gamma \vdash A\ \mathit{true})$
- $t[\sigma] :: (\Gamma \vdash J)$ is defined as follows, by analyzing t:

$$
\begin{array}{rcl}
(p[\sigma_0])[\sigma] & = & p[\sigma_0[\sigma]] \\
(\mathsf{con}(\varphi))[\sigma] & = & \mathsf{con}(p \mapsto \varphi(p)[\sigma]) \\
(\mathsf{sub}(\rho))[\sigma] & = & \mathsf{sub}(\kappa \mapsto \rho(\kappa)[\sigma]) \\
(\kappa\, V)[\sigma] & = & \begin{cases} K \bullet V[\sigma] & \rho(\kappa) = K \\ \kappa\,(V[\sigma]) & \kappa \notin \mathrm{dom}(\rho) \end{cases} \\
 & & \text{where } \sigma = \mathsf{sub}(\rho)
\end{array}
$$

The identity principles are analogous to η-expansion, cut principles to β-reduction. Operationally, Id_κ is the *identity continuation,* which takes in an A-value, reconstructs it and passes it to κ, while *id* denotes the *identity substitution.* The cut $K \bullet V$ denotes the result of applying K to V, while the cut $t[\sigma]$ denotes the result of substituting σ in t.

Do these definitions make sense? It is not difficult to see that both the identity and cut derivations necessarily terminate if the subformula ordering $\prec$ is well-founded, which indeed it is for the patterns we defined above. However, in general, for arbitrary recursive types, the subformula ordering will not in general be well-founded. Instead, we adopt the conventions of Girard (2001):

1. Reading the four inference rules for canonical derivations *coinductively,* we can verify that Id_κ and *id* are productive.
2. However, the definitions of $K \bullet V$ and $t[\sigma]$ are *not* necessarily productive, and so we must allow the possibility that cut-elimination yields divergence.

It is important to note that although the second convention allows expressions to be partial, it is still the case that values, continuations, and substitutions remain fully-defined, up to their embedded expressions. We write $\Omega :: (\Gamma \vdash \#)$ to denote the diverging expression.

In fact, for the sake of building examples and counterexamples, we will find it useful to include one additional effect besides nontermination:

$$\frac{}{\Gamma \vdash \#}\ \mho$$

The rule $\mho$ is pronounced "wrong".[3] Intuitively, $\mho$ can be thought of as some bad state (e.g., a pattern-matching exception raised by a missing branch) that refinement typing will rule out. The purpose of Ω and $\mho$ will be made more clear in §3.

Finally, we can relate the identity and cut terms by a lemma:

Lemma 8 (Identity composition)**.** *For all $t :: (\Gamma, \Delta \vdash J)$ and $V :: (\Gamma \vdash A\ \mathit{true})$ and $\sigma :: (\Gamma \vdash \Delta)$:*

1. *$t[id] = t$*
2. *$Id_\kappa \bullet V = \kappa\, V$*
3. *$id[\sigma] = \sigma$*

(For the obvious notion of extensional equality.)

Proof. We prove the identities by mutual coinduction. (1) reduces to (2) in the case $(\kappa\, V)[id] = Id_\kappa \bullet V = \kappa\, V$. (2) reduces to (3) by $Id_\kappa \bullet p[\sigma] = (\kappa\,(p[id]))[\sigma] = \kappa\,(p[id[\sigma]]) = \kappa\,(p[\sigma])$. (3) reduces to (1) as follows: Substitutions are equal iff they map continuation variables to equal continuations, and continuations are equal iff they map patterns to equal expressions. For all κ and p we have $(id[\sigma])(\kappa)(p) = (Id_\kappa[\sigma])(p) = (\kappa\,(p[id]))[\sigma] = \sigma(\kappa) \bullet p[id] = \sigma(\kappa)(p)[id] = \sigma(\kappa)(p)$, with the last step by (1). □

2.5 An embedding in Agda

Both the syntax of §2.2–§2.3 and the semantics of §2.4 are *intrinsic* (or "Church-style"): they consider only well-typed terms (as terms are well-typed by definition, corresponding to focusing proofs). In order to make this interpretation more concrete, as well as clarify some issues and give examples, in this section we will sketch how the language may be embedded into Agda, a total functional programming language based on dependent type theory. Agda's syntax is Haskell-like, and should hopefully be intelligible to someone familiar with other dependently-typed languages such as Coq, but the reader might skim or skip this section on first reading. The entire development of this paper is available online.[4]

After a simple prelude, the first thing we do is define the grammar of types `Tp`. In addition to the connectives described above, we include, for illustration, datatypes `nat` for natural numbers, and the higher-order datatype $\mathtt{dom} = \mathtt{nat} \oplus {}^{v}\!\to\!\mathtt{dom}$. (No doubt something could be learned by encoding recursive types more abstractly, but we forgo this for the sake of keeping the formalization concrete.) Next come hypotheses and contexts (and de Bruijn indices):

```
data Hyp : Set where
  FalseH : Tp -> Hyp
Ctx = List Hyp
data _∈_ : Hyp -> Ctx -> Set where
  f0 : {h : Hyp} {Δ : Ctx}
       -> h ∈ (h :: Δ)
  fS : {h h' : Hyp} {Δ : Ctx}
       -> h' ∈ Δ -> h' ∈ (h :: Δ)
```

We then define patterns as an inductive family:

[3] Girard calls this "daimon", written ✠. We borrow the more symmetric notation $\mho$ from Melliès and Vouillon (2005).

[4] `http://www.cs.cmu.edu/~noam/refinements/`

```
data _⊩_ : Ctx -> Tp -> Set where
   wild : forall {A} -> ( FalseH A :: []) ⊩ ¬ A
   #<> : [] ⊩ unit
   #inl : forall {Δ A B}
     -> Δ ⊩ A -> Δ ⊩ A + B
   #inr : forall {Δ A B}
     -> Δ ⊩ B -> Δ ⊩ A + B
   #pair : forall {Δ1 Δ2 A B}
     -> Δ1 ⊩ A -> Δ2 ⊩ B -> Δ1 ++ Δ2 ⊩ A * B
   #z : [] ⊩ nat
   #s : forall {Δ}
     -> Δ ⊩ nat -> Δ ⊩ nat
   #dn : forall {Δ}
     -> Δ ⊩ nat -> Δ ⊩ dom
   #dk : forall {Δ}
     -> Δ ⊩ ¬ dom -> Δ ⊩ dom
```

Note that variables within curly braces represent implicit parameters in Agda, which the typechecker attempts to infer. For example, the `pair` constructor takes two explicit arguments `p1 : Δ1 ⊩ A` and `p2 : Δ2 ⊩ B`, while `Δ1`, `Δ2`, `A`, and `B` are inferred. Finally, we can define focusing proofs:

```
data Judge : Set where
  True : Tp -> Judge
  False : Tp -> Judge
  All : Ctx -> Judge
  Contra : Judge

LCtx = List Ctx
data _∈∈_ : Hyp -> LCtx -> Set where
  s0 : {h : Hyp} {Δ : Ctx} {Γ : LCtx}
       -> h ∈ Δ -> h ∈∈ (Δ :: Γ)
  sS : {h : Hyp} {Δ : Ctx} {Γ : LCtx}
       -> h ∈∈ Γ -> h ∈∈ (Δ :: Γ)

codata _⊢_ : LCtx -> Judge -> Set where
  -- values
  _[_] : forall {Γ A Δ}
    -> Δ ⊩ A -> Γ ⊢ All Δ
    -> Γ ⊢ True A

  -- continuations
  con : forall {Γ A}
    -> (forall {Δ}
         -> (Δ ⊩ A) -> Δ :: Γ ⊢ Contra)
    -> Γ ⊢ False A

  -- substitutions
  sub : forall {Γ Δ}
    -> (forall {A}
         -> FalseH A ∈ Δ -> Γ ⊢ False A)
    -> Γ ⊢ All Δ

  -- expressions
  _∘_ : forall {Γ A}
    -> FalseH A ∈∈ Γ -> Γ ⊢ True A
    -> Γ ⊢ Contra

  ℧ : forall {Γ} -> Γ ⊢ Contra  -- daimon/wrong
```

Observe the use of the `codata` keyword marking the definition as coinductive. Now we can define the identity terms exactly as in §2.4, modulo some de Bruijn index reasoning (note that the `~` symbol is used to define corecursive functions in Agda):

```
mutual
 IdCon : forall {Γ A}
     -> (FalseH A ∈∈ Γ) -> Γ ⊢ False A
 IdCon κ ~ con(\p -> (sS κ) ∘ (p [ IdSub ]))
 IdSub : forall {Γ Δ}
     -> Δ :: Γ ⊢ All Δ
 IdSub ~ sub(\κ -> IdCon (s0 κ))
```

Agda verifies that these definitions are productive. In formal detail, the proof of cut is slightly more involved than was suggested above, since as is typical with such proofs, one must generalize the induction hypothesis to allow substitution for a context anywhere in the context list, and also prove a weakening lemma. After defining an inductive family `split Γ Δ Γ'` which encodes that `Γ'` splits into a context list `Γ` and a context `Δ` (with constants `sp0` and `spS`), we prove the following easy lemmas:

```
split< : forall {Γ Δ Γ' h}
  -> split Γ Δ Γ' -> h ∈∈ Γ'
  -> Either (h ∈ Δ) (h ∈∈ Γ)

weaken : forall {Γ Δ Γ' J}
     -> split Γ Δ Γ' -> Γ ⊢ J -> Γ' ⊢ J
```

Now we can state and prove cut, essentially as in §2.4, but just slightly more rigorously:

```
mutual
 Cut• : forall {Γ A}
    -> Γ ⊢ False A -> Γ ⊢ True A -> Γ ⊢ Contra
 Cut• (con φ) (p [ σ ]) ~ Cut[] sp0 (φ p) σ
 Cut[] : forall {Γ Δ Γ' J}
     -> split Γ Δ Γ'
     -> Γ' ⊢ J -> Γ ⊢ All Δ -> Γ ⊢ J
 Cut[] w (p [ σ0 ]) σ ~ p [ Cut[] w σ0 σ ]
 Cut[] w (con φ) σ ~ con (\p -> Cut[] (spS w) (φ p)
                                (weaken sp0 σ))
 Cut[] w (sub ρ) σ ~ sub (\κ -> Cut[] w (ρ κ) σ)
 Cut[] w (κ ∘ V) σ with split< w κ | σ
 ...  | Left i | sub ρ ~ Cut• (ρ i) (Cut[] w V σ)
 ...  | Right κ' | _ ~ κ' ∘ (Cut[] w V σ)
 Cut[] w ℧ σ ~ ℧
```

Here Agda rightfully complains that `Cut•` and `Cut[]` are not productive, though it accepts the definition. Having irritated Agda once, we may as well do so again with the following definition:

```
Ω : forall {Γ} -> Γ ⊢ Contra
Ω ~ Ω
```

It is worth pointing out that in all of the terms we defined above and which we will define below, types are mentioned only in signatures, not in the actual bodies of terms—even though we are programming à la Church. This is a consequence of the subformula property, that types are always inferable from context.

EXAMPLE 1. We define addition (as a continuation transformer):

```
plus : forall {Γ}
   -> (FalseH nat ∈∈ Γ) -> Γ ⊢ False (nat * nat)
plus κ = con (\p -> (sS κ) ∘ (plus* p [ IdSub ]))
  where
    plus* : forall {Δ} -> Δ ⊩ nat * nat -> Δ ⊩ nat
    plus* (#pair #z n) = n
    plus* (#pair (#s m) n) = #s (plus* (#pair m n))
```

Here we are using the full power of the higher-order representation, defining the continuation `plus` in terms of the auxiliary meta-function `plus*`, which adds two `nat`-patterns. ■

EXAMPLE 2. We define a continuation on $\texttt{dom} = \texttt{nat} \oplus \overline{\nu}\texttt{dom}$ which attempts to apply its argument to zero:

```
app0 : forall {Γ} -> Γ ⊢ False dom
app0 = con app0*
  where
    app0* : forall {Δ} -> Δ ⊩ dom -> _ ⊢ Contra
    app0* (#dk wild) = (s0 f0) ∘ (#dn #z [ Sub0 ])
    app0* _ = ℧
```

Here `Sub0` stands for the empty substitution. Note that without the extra case "`app0* _ =` $\mho$", Agda will not accept the definition. ■

Both examples illustrate classic applications for refinement types. Beyond the fact that `plus` takes a pair of `nat`s and returns a `nat`, we might for instance like to know that it takes a pair of even `nat`s to another even `nat`, a pair of odd `nat`s to an even `nat`, etc. Likewise, when we write `app0` we have in mind that it will only be called with an argument that falls under the $\overset{v}{\neg}$`dom` branch of the `dom` datatype, but this invariant is not captured by its type. The compiler cannot rule out the senseless pattern branch unless we refine the type of `app0` to explicitly state our invariant.

3. Refining focusing proofs

Above we described an intrinsic Curry-Howard interpretation of focusing proofs as well-typed terms of a language with recursive datatypes and continuations. In this section, we will give an *extrinsic* view of refinement-typing *of those well-typed terms.* This kind of two-layer approach originates with Freeman and Pfenning (1991), who introduced refinements as a way of verifying additional properties of already well-typed programs, rather than for defining new programs.[5]

3.1 Refining types

In order to check additional properties of programs, we must be able to specify these properties. Following Pfenning (2008), we will try to avoid confusion between ordinary types A and refinement properties $S \sqsubseteq A$ of those types by calling the latter *sorts,* though we will still refer to the process of verifying sorts of terms as refinement typing. A sort $S \sqsubseteq A$ is *not* a subtype of A in the traditional sense. For example, while it will be true (by definition) that every value of sort A has type A, it will also be true (by definition) that every continuation accepting sort S accepts type A.

Definition 9 (Refinement contexts)**.** *A* **refinement context** $\Psi \sqsubseteq \Delta$ *is a mapping from continuation variables* $\kappa :: (A\ \textit{false} \in \Delta)$ *to a set of sorts refining* A*.*

For notational convenience, we will sometimes write a refinement context as a simple list of hypotheses $\kappa_1 \Leftarrow S_1, \ldots, \kappa_n \Leftarrow S_n$, where the same κ can appear any number of times. We write $\kappa \Leftarrow S$ rather than $\kappa : S$ to emphasize that κ stands for a continuation *accepting* sort S, not a value of sort S. The set $\Psi(\kappa)$ is interpreted *conjunctively,* i.e., we assume that κ accepts all of the sorts $S \in \Psi(\kappa)$ (but note that because of the contravariant reading, this is the same as saying that it accepts the union sort $\bigcup \Psi(\kappa)$).

It will be useful to have several operations on refinement contexts. Let $\Psi_1 \sqsubseteq \Delta_1$ and $\Psi_2 \sqsubseteq \Delta_2$. Their *concatenation* $(\Psi_1, \Psi_2) \sqsubseteq (\Delta_1, \Delta_2)$ is defined by

$$(\Psi_1, \Psi_2)(\kappa) = \begin{cases} \Psi_1(\kappa) & \text{if } \kappa \in \mathrm{dom}(\Delta_1) \\ \Psi_2(\kappa) & \text{if } \kappa \in \mathrm{dom}(\Delta_2) \end{cases}$$

The unit of concatenation is $\cdot \sqsubseteq \cdot$ (which denotes the empty map).

[5] It is worth noting, though, that historically, *all* approaches to intersection and union types have been two-layer in the sense that they begin by defining untyped terms, and then define typing for those terms. The difference is that we start with a typed rather than an untyped syntax. But this is simply a *generalization* of the traditional approach, because (as the well-known pun goes) untyped syntax is really "uni-typed". Indeed, because we defined both syntax and semantics generically with respect to patterns, the untyped story is literally what we get by taking all patterns to introduce a single, universal type, without any modification to the formalism. The benefit we derive from starting with a (multi-)typed syntax is mainly conceptual, because it has a direct logical interpretation and so is easier to reason about.

$$\frac{}{\top \sqsubseteq A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq A}{S \cap T \sqsubseteq A}$$

$$\frac{}{\bot \sqsubseteq A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq A}{S \cup T \sqsubseteq A}$$

$$\frac{S \sqsubseteq A}{\overset{v}{\neg} S \sqsubseteq \overset{v}{\neg} A} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq B}{S \otimes T \sqsubseteq A \otimes B} \qquad \frac{S \sqsubseteq A \quad T \sqsubseteq B}{S \oplus T \sqsubseteq A \oplus B}$$

Figure 2. Some sort constructors

Let $\Psi_1, \Psi_2 \sqsubseteq \Delta$. Their *meet* $(\Psi_1 \wedge \Psi_2) \sqsubseteq \Delta$ is defined by

$$(\Psi_1 \wedge \Psi_2)(\kappa) = \Psi_1(\kappa) \cup \Psi_2(\kappa)$$

The unit for the meet is $\mathbb{T} \sqsubseteq \Delta$ (defined by $\mathbb{T}(\kappa) = \emptyset$).

Just as the meaning of types was defined by their patterns, the meaning of sorts is defined by *pattern-inversion.* To define the sort $S \sqsubseteq A$ one specifies, for every pattern $p :: (\Delta \Vdash A\ \textit{true})$, a set $[\![p : S]\!]$ of refinement contexts of Δ. Intuitively, the set $[\![p : S]\!]$ is interpreted *disjunctively,* as the different possible refinements of the variables bound by the pattern, assuming it has the given sort. For example, intersection and union sorts are defined by:

$$\begin{aligned} [\![p : S \cap T]\!] &= \{(\Psi_1 \wedge \Psi_2) \mid \Psi_1 \in [\![p : S]\!], \Psi_2 \in [\![p : T]\!]\} \\ [\![p : S \cup T]\!] &= [\![p : S]\!] \cup [\![p : T]\!] \\ [\![p : \top]\!] &= \{\mathbb{T}\} \\ [\![p : \bot]\!] &= \emptyset \end{aligned}$$

... and congruence refinements for the type constructors by:

$$\begin{aligned} [\![\kappa : \overset{v}{\neg} S]\!] &= \{(\kappa \Leftarrow S)\} \\ [\![(p_1, p_2) : S \otimes T]\!] &= \{(\Psi_1, \Psi_2) \mid \Psi_1 \in [\![p_1 : S]\!], \Psi_2 \in [\![p_2 : T]\!]\} \\ [\![\mathsf{inl}\, p : S \oplus T]\!] &= [\![p : S]\!] \\ [\![\mathsf{inr}\, p : S \oplus T]\!] &= [\![p : T]\!] \end{aligned}$$

For these definitions to make sense, we have to respect some implicit conventions on the formation of sorts. For example, the binary union and intersection can only be formed when both S and T refine the same type. These implicit conventions are described in Figure 2. In addition to these generic refinement constructors, we can define some more interesting refinements of datatypes. Here are a few refinements of `nat`:

$$[\![\mathtt{z} : \mathtt{even}]\!] = \{\cdot\} \qquad [\![\mathtt{z} : \mathtt{odd}]\!] = [\![\mathtt{z} : \mathtt{pos}]\!] = \emptyset$$

$$[\![\mathtt{s}\, p : \mathtt{even}]\!] = [\![p : \mathtt{odd}]\!] \qquad [\![\mathtt{s}\, p : \mathtt{odd}]\!] = [\![p : \mathtt{even}]\!]$$

$$[\![\mathtt{s}\, p : \mathtt{pos}]\!] = [\![p : \top]\!]$$

And a couple refinements of `dom` (recall $\mathtt{dom} = \mathtt{nat} \oplus \overset{v}{\neg}\mathtt{dom}$):

$$[\![\mathtt{dn}\, p : \mathtt{dom_{even}}]\!] = [\![p : \mathtt{even}]\!] \qquad [\![\mathtt{dn}\, p : \mathtt{dom^*}]\!] = \emptyset$$

$$[\![\mathtt{dk}\, p : \mathtt{dom_{even}}]\!] = [\![p : \overset{v}{\neg}\mathtt{dom_{even}}]\!] \qquad [\![\mathtt{dk}\, p : \mathtt{dom^*}]\!] = [\![p : \overset{v}{\neg}\top]\!]$$

The reader can try working out the following examples:

$$\begin{aligned} [\![(\kappa_1, \kappa_2) : \overset{v}{\neg}\mathtt{even} \otimes \overset{v}{\neg}\mathtt{odd}]\!] &= \{(\kappa_1 \Leftarrow \mathtt{even}, \kappa_2 \Leftarrow \mathtt{odd})\} \\ [\![\kappa : \overset{v}{\neg}\mathtt{even} \cap \overset{v}{\neg}\mathtt{odd}]\!] &= \{(\kappa \Leftarrow \mathtt{even}, \kappa \Leftarrow \mathtt{odd})\} \\ [\![\kappa : \overset{v}{\neg}\mathtt{even} \cup \overset{v}{\neg}\mathtt{odd}]\!] &= \{(\kappa \Leftarrow \mathtt{even}), (\kappa \Leftarrow \mathtt{odd})\} \\ [\![\mathtt{z} : \mathtt{even} \cap \mathtt{odd}]\!] &= \emptyset \\ [\![\mathtt{z} : \mathtt{even} \cup \mathtt{odd}]\!] &= \{\cdot\} \end{aligned}$$

3.2 Refining terms

Just as focusing proofs were defined generically by reference to patterns—without mentioning any particular types—refinement typing is defined generically by reference to pattern-inversion. The refinement typing judgment takes the form $\Xi \vDash t : \mathcal{J}$, where $t :: (\Gamma \vdash J)$ is a term, $\mathcal{J}$ refines J, and Ξ refines the context list Γ

Refinement contexts
$\Psi \sqsubseteq \Delta$ maps every $\kappa :: (A\ \mathit{false} \in \Delta)$ to a set of $S \sqsubseteq A$
$\Xi \sqsubseteq \Gamma$ maps every $\kappa :: (A\ \mathit{false} \in \Gamma)$ to a set of $S \sqsubseteq A$
Refinement judgments
$t{:}\mathcal{J} \quad ::= \quad V : S \mid K \Leftarrow S \mid \sigma : \Delta \mid E : \#$

$$\frac{\Psi \in [\![p : S]\!] \quad \Xi \vDash \sigma : \Psi}{\Xi \vDash p[\sigma] : S} \qquad \frac{\Psi \in [\![p : S]\!] \quad \longrightarrow \quad \Xi, \Psi \vDash \varphi(p) : \#}{\Xi \vDash \mathsf{con}(\varphi) \Leftarrow S}$$

$$\frac{S \in \Psi(\kappa) \quad \longrightarrow \quad \Xi \vDash \rho(\kappa) \Leftarrow S}{\Xi \vDash \mathsf{sub}(\rho) : \Psi} \qquad \frac{S \in \Gamma(\kappa) \quad \Gamma \vDash V : S}{\Gamma \vDash \kappa\, V : \#}$$

(no rule for $\mho$)

Figure 3. Refinement typing of focusing proofs

(which means that it sends any $\kappa :: (A\ \mathit{false} \in \Gamma)$, to a set $\Xi(\kappa)$ of sorts of A). We adopt the same notational convention for Ξs as we did for Ψs.

A value $p[\sigma]$ has sort S if there is some Ψ in $[\![p : S]\!]$ such that σ satisfies all of Ψ:

$$\frac{\Psi \in [\![p : S]\!] \quad \Xi \vDash \sigma : \Psi}{\Xi \vDash p[\sigma] : S}$$

Note that implicit in this rule are the conditions:

$$p :: (\Delta \Vdash A\ \mathit{true}) \qquad \sigma :: (\Gamma \vdash \Delta)$$

$$\Xi \sqsubseteq \Gamma \qquad \Psi \sqsubseteq \Delta \qquad S \sqsubseteq A$$

But it is safe to *leave* these conditions implicit, since they are the only sensical way to interpret the rule, by our definition of patterns, substitutions, and pattern-inversion.

A continuation $\mathsf{con}(\varphi)$ accepts sort $S \sqsubseteq A$ if for every A-pattern p, in every possible context $\Psi \in [\![p : S]\!]$, the expression $\varphi(p)$ is well-sorted:

$$\frac{\Psi \in [\![p : S]\!] \quad \longrightarrow \quad \Xi, \Psi \vDash \varphi(p) : \#}{\Xi \vDash \mathsf{con}(\varphi) \Leftarrow S}$$

Again, this rule leaves implicit various conditions that are forced for the rule to make sense. Notice that if S is a union, checking the continuation could involve checking the same branch $\varphi(p)$ within multiple refinement contexts.

A substitution $\mathsf{sub}(\rho)$ satisfies all of $\Psi \sqsubseteq \Delta$ if for every continuation variable $\kappa :: (A\ \mathit{false} \in \Delta)$, for every hypothesis $S \in \Psi(\kappa)$, the continuation $\rho(\kappa)$ accepts sort S:

$$\frac{S \in \Psi(\kappa) \quad \longrightarrow \quad \Xi \vDash \rho(\kappa) \Leftarrow S}{\Xi \vDash \mathsf{sub}(\rho) : \Psi}$$

Notice that if Ψ comes from inverting a pattern at an intersection, checking the substitution could involve checking the same continuation $\rho(\kappa)$ against multiple sorts.

An expression $\kappa\, V$ is well-sorted in context Ξ if there is at least some hypothesis $S \in \Xi(\kappa)$ such that V has sort S:

$$\frac{S \in \Xi(\kappa) \quad \Xi \vDash V : S}{\Xi \vDash \kappa\, V : \#}$$

Finally, there is *no* rule that makes the expression $\mho$ well-sorted, which brings us to the following important slogan:

well-sorted programs don't go $\mho$

The complete set of rules for refinement typing are listed for reference in Figure 3.

3.3 Refining cut and identity

The technical meaning of the slogan above is contained in the sort preservation theorem:

Theorem 10 (Sort preservation of cuts)**.**

1. *If* $\Xi \vDash K \Leftarrow S$ *and* $\Xi \vDash V : S$ *then* $\Xi \vDash K \bullet V : \#$
2. *If* $\Xi \vDash \sigma : \Psi$ *and* $\Xi, \Psi \vdash t : \mathcal{J}$ *then* $\Xi, \Psi \vdash t[\sigma] : \mathcal{J}$

The proof of sort preservation exactly mirrors the structure of cut-elimination, since the rules for refinement typing precisely mirror the rules for building focusing proofs. Again, we must adopt a coinductive interpretation of refinement typing, and the convention that the diverging expression Ω is always well-sorted.

Similarly, the proof of identity can be reflected back to sorts:

Theorem 11 (Sort identity)**.**

1. *If* $S \in \Xi(\kappa)$ *then* $\Xi \vDash \mathit{Id}_\kappa \Leftarrow S$
2. $\Xi, \Psi \vDash \mathit{id} : \Psi$

3.4 Subsorting: the identity coercion interpretation

In fact, the identity terms Id_κ and id are much more interesting in the setting of refinement typing, since they give us a coercion interpretation of *subsorting*.

Definition 12 (Subsorting)**.** *Let* $S, T \sqsubseteq A$*. We say that* S *is a* **subsort** *of* T *if* $\kappa \Leftarrow T \vDash \mathit{Id}_\kappa \Leftarrow S$*. The subsorting relationship is written* $S \leq_A T$*, or just* $S \leq T$ *leaving* A *implicit. We write* $S \equiv T$ *if* $S \leq T$ *and* $T \leq S$*, and* $S \not\leq T$ *if not* $S \leq T$*.*

Intuitively, $S \leq T$ says that we can uniformly convert any T-continuation into an S-continuation by precomposing it with the identity—hence the *identity coercion interpretation.* We also have an equivalent interpretation using the identity substitution *id*:

Definition 13. *Let* $\Psi, \Psi' \sqsubseteq \Delta$*. We say that* Ψ *is a* **subcontext** *of* Ψ' *(written* $\Psi \leq_\Delta \Psi'$*) if* $\Psi \vDash \mathit{id} : \Psi'$*.*

Proposition 14. $S \leq_A T$ *iff for all* A*-patterns* p*, for every* $\Psi \in [\![p : S]\!]$ *there exists* $\Psi' \in [\![p : T]\!]$ *such that* $\Psi \leq \Psi'$*.*

Proof. Immediate by definition of the identity terms. □

Proposition 15. *Both* $\leq_A$ *and* $\leq_\Delta$ *are reflexive and transitive.*

Proof. Reflexivity is immediate by sort identity, while transitivity follows from sort preservation combined with the identity composition lemma. □

Proposition 16 (Term inclusion/reverse inclusion)**.**

- *If* $\Xi \vDash V : S$ *and* $S \leq T$ *then* $\Xi \vDash V : T$
- *If* $\Xi \vDash K \Leftarrow T$ *and* $S \leq T$ *then* $\Xi \vDash K \Leftarrow S$
- *If* $\Xi \vDash \sigma : \Psi$ *and* $\Psi \leq \Psi'$ *then* $\Xi \vDash \sigma : \Psi'$
- *If* $\Xi, \Psi' \vDash E : \#$ *and* $\Psi \leq \Psi'$ *then* $\Xi, \Psi \vDash E : \#$

Proof. All consequences of the identity composition lemma. □

Proposition 17. $\leq_A$ *is a distributive lattice with* $\cap$*,* $\cup$*,* $\top$*, and* $\bot$*.*

Proposition 18. $\oplus, \otimes$*, and* $\overset{v}{\neg}$ *obey the usual covariant and contravariant laws:*

1. *If* $S_1 \leq T_1$ *and* $S_2 \leq T_2$ *then* $S_1 \otimes T_1 \leq S_2 \otimes T_2$
2. *If* $S_1 \leq T_1$ *and* $S_2 \leq T_2$ *then* $S_1 \oplus T_1 \leq S_2 \oplus T_2$
3. *If* $T \leq S$ *then* $\overset{v}{\neg} S \leq \overset{v}{\neg} T$

Proposition 19. *Intersections and unions distribute through sums and products, as follows:*

1. $S \otimes \bot \equiv \bot$

2. $S \otimes (T_1 \cap T_2) \equiv (S \otimes T_1) \cap (S \otimes T_2)$
3. $S \oplus (T_1 \cap T_2) \equiv (S \oplus T_1) \cap (S \oplus T_2)$
4. $S \otimes (T_1 \cup T_2) \equiv (S \otimes T_1) \cup (S \otimes T_2)$
5. $S \oplus (T_1 \cup T_2) \equiv (S \oplus T_1) \cup (S \oplus T_2)$

Proof (of Props. 17–19). Immediate by Prop. 14 and the pattern-inversion rules. □

Besides these typical properties, more interesting are the laws and non-laws we can derive for distributing intersections and unions through call-by-value negation:

Proposition 20.

1. *(i)* $\top \equiv {}^{v}\neg\bot$ *and (ii)* ${}^{v}\neg S \cap {}^{v}\neg T \equiv {}^{v}\neg(S \cup T)$
2. *(i)* ${}^{v}\neg\top \not\leq \bot$ *and (ii) in general* ${}^{v}\neg(S \cap T) \not\leq {}^{v}\neg S \cup {}^{v}\neg T$

Proof. 1(i) reduces to checking $\cdot \vDash Id_\kappa \Leftarrow \bot$, which holds vacuously since $[\![p : \bot]\!] = \emptyset$. 1(ii) reduces to checking $\kappa \Leftarrow S, \kappa \Leftarrow T \vDash Id_\kappa \Leftarrow S \cup T$. Since $[\![p : S \cup T]\!] = [\![p : S]\!] \cup [\![p : T]\!]$, this reduces to checking that Id_κ accepts both S and T, and that holds by sort identity.

2(i) fails because $[\![p : \bot]\!]$ is empty but $[\![p : {}^{v}\neg\top]\!]$ is not. 2(ii) requires that either $\kappa \Leftarrow S \cap T \vDash Id_\kappa \Leftarrow S$ or $\kappa \Leftarrow S \cap T \vDash Id_\kappa \Leftarrow T$, but these hold only if $S \leq T$ or $T \leq S$. □

3.5 Revisiting the value/evaluation context restrictions

In this section we explain in what sense the value and evaluation context restrictions, as well as the failures of subtyping described in the introduction, are already built into focusing proofs. Let us fix a type A, and sorts $S, T \sqsubseteq A$. We begin by observing that the following rules are admissible:

$$\frac{\Xi \vDash V : S \quad \Xi \vDash V : T}{\Xi \vDash V : S \cap T} \qquad \frac{\Xi \vDash V : S}{\Xi \vDash V : S \cup T} \qquad \frac{\Xi \vDash V : T}{\Xi \vDash V : S \cup T}$$

The value restriction comes by default in these rules, in the trivial sense that they are rules for typing values. But how does this relate to the value restriction (required for intersections but not for unions) in ML? In general, arbitrary terms of call-by-value λ-calculus are embedded in the focusing language by CPS translation. We can translate ML terms either as expressions in a context with a distinguished continuation variable accepting type A, or (slightly less directly) as values of double-negated type ${}^{v}\neg{}^{v}\neg A$. For simplicity let's take the latter interpretation. Thus the usual unrestricted union-introduction rules correspond to the following admissible rules for checking values of double-negated type:

$$\frac{\Xi \vDash V : {}^{v}\neg{}^{v}\neg S}{\Xi \vDash V : {}^{v}\neg{}^{v}\neg(S \cup T)} \qquad \frac{\Xi \vDash V : {}^{v}\neg{}^{v}\neg T}{\Xi \vDash V : {}^{v}\neg{}^{v}\neg(S \cup T)}$$

On the other hand, the unrestricted intersection-introduction rule corresponds to the following:

$$\frac{\Xi \vDash V : {}^{v}\neg{}^{v}\neg S \quad \Xi \vDash V : {}^{v}\neg{}^{v}\neg T}{\Xi \vDash V : {}^{v}\neg{}^{v}\neg(S \cap T)} \ *$$

which is *not* admissible. Similarly, we can observe that the following rules are admissible:

$$\frac{\Xi \vDash K \Leftarrow S}{\Xi \vDash K \Leftarrow S \cap T} \qquad \frac{\Xi \vDash K \Leftarrow T}{\Xi \vDash K \Leftarrow S \cap T} \qquad \frac{\Xi \vDash K \Leftarrow S \quad \Xi \vDash K \Leftarrow T}{\Xi \vDash K \Leftarrow S \cup T}$$

These rules have an evaluation context restriction by default, in the sense that they check *call-by-value* continuations, and call-by-value continuations are isomorphic to evaluation contexts.[6] This time, we can lift intersections through double-negation:

$$\frac{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg S}{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg(S \cap T)} \qquad \frac{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg T}{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg(S \cap T)}$$

But we *cannot* do so for unions:

$$\frac{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg S \quad \Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg T}{\Xi \vDash K \Leftarrow {}^{v}\neg{}^{v}\neg(S \cup T)} \ *$$

By the inclusion/reverse inclusion lemma, these facts about refinement typing reduce to the following properties of subsorting:

Proposition 21.

1. *If* $S \leq T$ *then* ${}^{v}\neg{}^{v}\neg S \leq {}^{v}\neg{}^{v}\neg T$
2. *In general* ${}^{v}\neg{}^{v}\neg S \cap {}^{v}\neg{}^{v}\neg T \not\leq {}^{v}\neg{}^{v}\neg(S \cap T)$
3. *In general* ${}^{v}\neg{}^{v}\neg(S \cup T) \not\leq {}^{v}\neg{}^{v}\neg S \cup {}^{v}\neg{}^{v}\neg T$

Proof. (1) reduces to Prop. 18.3, while (2) and (3) reduce to the failure of Prop. 20.2(ii), since

$${}^{v}\neg{}^{v}\neg S \cap {}^{v}\neg{}^{v}\neg T \equiv {}^{v}\neg({}^{v}\neg S \cup {}^{v}\neg T) \not\leq {}^{v}\neg{}^{v}\neg(S \cap T)$$
$${}^{v}\neg{}^{v}\neg(S \cup T) \equiv {}^{v}\neg({}^{v}\neg S \cap {}^{v}\neg T) \not\leq {}^{v}\neg{}^{v}\neg S \cup {}^{v}\neg{}^{v}\neg T$$

□

Finally, we can consider the subtyping laws for call-by-value functions by the encoding $A \xrightarrow{v} B = {}^{v}\neg(A \otimes {}^{v}\neg B)$. We have that

$$\begin{aligned}(A \xrightarrow{v} C) \cap (B \xrightarrow{v} C) &= {}^{v}\neg(A \otimes {}^{v}\neg C) \cap {}^{v}\neg(B \otimes {}^{v}\neg C)\\ &\equiv {}^{v}\neg((A \otimes {}^{v}\neg C) \cup (B \otimes {}^{v}\neg C))\\ &\equiv {}^{v}\neg((A \cup B) \otimes {}^{v}\neg C)\\ &= (A \cup B) \xrightarrow{v} C\end{aligned}$$

but on the other hand

$$\begin{aligned}(A \xrightarrow{v} B) \cap (A \xrightarrow{v} C) &= {}^{v}\neg(A \otimes {}^{v}\neg B) \cap {}^{v}\neg(A \otimes {}^{v}\neg C)\\ &\equiv {}^{v}\neg((A \otimes {}^{v}\neg B) \cup (A \otimes {}^{v}\neg C))\\ &\equiv {}^{v}\neg(A \otimes ({}^{v}\neg B \cup {}^{v}\neg C))\\ &\not\leq {}^{v}\neg(A \otimes {}^{v}\neg(B \cap C))\\ &= A \xrightarrow{v} (B \cap C)\end{aligned}$$

3.6 Subsorting: the no-counterexamples interpretation

We have given a synthetic reconstruction of the value and evaluation context restrictions proposed by Davies and Pfenning (2000) and Dunfield and Pfenning (2004), within the logical setting of focusing. This is reassuring, because it suggests that focusing type systems can be used as a safe foundation for effectful languages. But it doesn't quite explain why omitting those restrictions leads to concrete safety *violations* in those languages. In this section we begin an attempt at answering that question, by considering another possible interpretation of subsorting.

Definition 22 (Safety)**.** *We say that the pair of a value* $V :: (\Gamma \vdash A \text{ true})$ *and continuation* $K :: (\Gamma \vdash A \text{ false})$ *are a* **safety violation** *(written* $V \not\perp K$*) if* $K \bullet V = \mho$*. We say that* V *and* K *are* **safe** *(written* $V \perp K$*) if they are not a safety violation.*

Note that $\perp$ is actually the complement of the *orthogonality* relation in ludics.[7]

Definition 23 (Safe subsorting)**.** *Let* $S, T \sqsubseteq A$*. We say that* S *is a* **safe subsort** *of* T *(written* $S \leqslant T$*) if* $V \perp K$ *for all closed* V *and* K *such that* $\vDash V : S$ *and* $\vDash K \Leftarrow T$*.*

We call this the *no-counterexamples interpretation* of subsorting. Clearly, if we have an explicit witness to the safety of a subsorting relationship using the identity coercion, then there can be no counterexamples.

[6] At least to a first approximation, cf. Danvy (2004).

[7] The idea of defining orthogonality as safety rather than as termination comes from Melliès and Vouillon (2005).

Theorem 24 (Soundness). *$S \leq T$ implies $S \leqslant T$*

Proof. By Prop. 16—either value inclusion or continuation reverse inclusion will do—combined with sort preservation. □

The really interesting question is completeness: if the identity coercion does not sortcheck, can we come up with an explicit safety violation, i.e.,

$$\textit{does } S \not\leq T \textit{ imply } S \not\leqslant T?$$

But whereas the identity coercion interpretation is fixed by the logical rules of focusing and the definitions of the connectives—and thereby open-ended with respect to language extension—the no-counterexamples interpretation is dependent upon the precise set of non-logical effects that there are in the language, besides Ω and $\mho$. In general, the question we need to ask is,

$$\textit{with what set of available effects, and for what types } A,\\ \textit{does } S \not\leq_A T \textit{ imply } S \not\leqslant_A T?$$

We will not attempt to give a full answer to this general question. However, in this section we will try to give a partial answer, by showing how the subtyping non-law Prop. 20.2(ii) is not *universally* safe: we exhibit a particular type ($^{v}\neg\mathtt{nat}$) at which a particular effect (non-determinism) is sufficient for building a counterexample (which can also be translated into counterexamples to all the other invalid subtyping laws and refinement typing rules of §3.5). In fact, the effects Ω and $\mho$ are already sufficient to build a counterexample to Prop. 20.2(i), at any type:

Notation. *If K is a continuation accepting type A, we write $\downarrow K$ to stand for the continuation treated as a value of type $^{v}\neg A$, i.e., $\downarrow K = \kappa[\mathsf{sub}(\kappa \mapsto K)]$.*

Proposition 25. $^{v}\neg\top \not\leqslant \bot$

Proof. A safety violation is provided by pairing the continuation which diverges on all inputs, treated as a value, together with the continuation that ignores its argument and returns $\mho$. Formally, $\downarrow \mathsf{con}(p \mapsto \Omega) \not\perp \mathsf{con}(\kappa \mapsto \mho)$. The left-hand value has sort $^{v}\neg\top$, while the right-hand continuation accepts sort $\bot$. □

Now we introduce non-determinism into the language.

Definition 26 (Choice). *For any pair of expressions E_1, E_2, we can form their* **erratic choice** *$E_1 \parallel E_2$, such that $E_1 \parallel E_2 = \mho$ if either $E_1 = \mho$ or $E_2 = \mho$. Erratic choice is sortchecked by:*

$$\frac{\Xi \vDash E_1 : \#\quad \Xi \vDash E_2 : \#}{\Xi \vDash E_1 \parallel E_2 : \#}$$

Proposition 27. *In the presence of erratic choice, the principle $^{v}\neg(S \cap T) \leqslant {}^{v}\neg S \cup {}^{v}\neg T$ fails at sorts $S = {}^{v}\neg\mathtt{even}$, $T = {}^{v}\neg\mathtt{odd}$.*

Proof. Let K_{01} be the $^{v}\neg\mathtt{nat}$-continuation which applies its argument to 0 or 1, non-deterministically using erratic choice:

$$K_{01} = \mathsf{con}(\kappa \mapsto \kappa\, 0 \parallel \kappa\, 1)$$

We have that $\vDash K_{01} \Leftarrow S \cap T$, although neither $\vDash K_{01} \Leftarrow S$ nor $\vDash K_{01} \Leftarrow T$. Hence $\vDash \downarrow K_{01} : {}^{v}\neg(S \cap T)$. Note that $\downarrow K_{01}$ is a $^{v}\neg{}^{v}\neg\mathtt{nat}$-value, which recall can be thought of as nat-returning computation. Now we construct a $^{v}\neg{}^{v}\neg\mathtt{nat}$-continuation K^* such that $\downarrow K_{01} \not\perp K^*$. Informally, we define K^* as the continuation which evaluates its argument twice, checks that the parity of the results is the same, and returns $\mho$ if not. Taking some liberties with syntactic sugar, we can write K^* like so:

$$\begin{aligned} K^* = \mathsf{con}(\ & \kappa \mapsto_\kappa \mathsf{con}(m \mapsto \\ & \kappa\, \mathsf{con}(n \mapsto \\ & \text{if } \mathrm{parity}(m) = \mathrm{parity}(n) \text{ then } \Omega \text{ else } \mho))) \end{aligned}$$

We have that $\vDash K^* \Leftarrow {}^{v}\neg S \cup {}^{v}\neg T$, although not $\vDash K^* \Leftarrow {}^{v}\neg(S \cap T)$. Since $\downarrow K_{01} \not\perp K^*$, we have the desired counterexample. □

3.7 Refinements in Agda

The foregoing prose description of a refinement type system for focusing proofs translates almost directly into Agda code. We will not go through its entirety, but only some choice snippets.

The development kicks off by defining a grammar of sort constructors, including the following:

```
data Sort : Tp -> Set where
  ⊤ : forall {A} -> Sort A
  ⊥ : forall {A} -> Sort A
  _∩_ : forall {A} -> Sort A -> Sort A -> Sort A
  _∪_ : forall {A} -> Sort A -> Sort A -> Sort A
  ¬r :  forall {A} -> Sort A -> Sort (¬ A)
  _*r_ : forall {A B} -> Sort A -> Sort B -> Sort (A * B)
  _+r_ : forall {A B} -> Sort A -> Sort B -> Sort (A + B)
```

as well as the datasorts `even`, `odd`, `pos`, `domeven`, and `dom*`. We define refinement hypotheses and contexts:

```
data RHyp : Hyp -> Set where
  ConH : {A : Tp} -> Sort A -> RHyp (FalseH A)
data RCtx (Δ : Ctx) : Set where
  RΔ : ({h : Hyp} -> h ∈ Δ -> List (RHyp h)) -> RCtx Δ
```

And then the various operations on refinement contexts (for which we list only types):

```
[]x : RCtx []
_++x_ : forall {Δ1 Δ2}
        -> RCtx Δ1 -> RCtx Δ2 -> RCtx (Δ1 ++ Δ2)
⊤x : forall {Δ} -> RCtx Δ
_∧x_ : forall {Δ} -> RCtx Δ -> RCtx Δ -> RCtx Δ
hyps : forall {Δ h} -> RCtx Δ -> h ∈ Δ -> List (RHyp h)
```

Now, the inversion function is defined by a big induction on patterns and sorts:

```
invert : forall {Δ A}
         -> (Δ ⊩ A) -> Sort A -> List (RCtx Δ)
invert p (S ∩ T) =
       concat (map (\Ψ1 -> map (\Ψ2 ->
       Ψ1 ∧x Ψ2) (invert p T)) (invert p S))
invert p (S ∪ T) = invert p S ++ invert p T
invert p ⊤ = ⊤x :: []
invert p ⊥ = []
-- (cases for ¬r, *r, and +r omitted)
invert #z even = []x :: []
invert (#s n) even =  invert n odd
-- (cases for odd, pos, dom, and domeven omitted)
```

We define refinement context lists `RLCtx Γ` with operations `[]xx`, `Ψ ::xx Ξ`, and `hyps' Ξ κ`, and then finally the coinductive rules of refinement typing:

```
data RJudge : Judge -> Set where
    RVal : {A : Tp} -> Sort A -> RJudge (True A)
    RCon : {A : Tp} -> Sort A -> RJudge (False A)
    RSub : {Δ : Ctx} -> RCtx Δ -> RJudge (All Δ)
    RExp : RJudge Contra

codata _⊨_is_ : forall {Γ J}
       -> RLCtx Γ -> (Γ ⊢ J) -> RJudge J -> Set where
  tpval : forall {Γ A Δ}
            {p : Δ ⊩ A} {σ : Γ ⊢ All Δ}
            {Ξ : RLCtx Γ} {S : Sort A} {Ψ : RCtx Δ}
    -> Ψ ∈ invert p S -> Ξ ⊨ σ is RSub Ψ
    -> Ξ ⊨ p [ σ ] is RVal S

  tpcon : forall {Γ A}
            {φ : forall {Δ}
```

```
          -> (Δ ⊩ A) -> Δ :: Γ ⊢ Contra}
         {Ξ : RLCtx Γ} {S : Sort A}
 -> (forall {Δ} {p : Δ ⊩ A} {Ψ : RCtx Δ}
     -> Ψ ∈ invert p S -> Ψ ::xx Ξ ⊨ φ(p) is RExp)
 -> Ξ ⊨ con(φ) is RCon S

tpsub : forall {Γ Δ}
         {ρ : forall {A}
            -> FalseH A ∈ Δ -> Γ ⊢ False A}
         {Ξ : RLCtx Γ} {Ψ : RCtx Δ}
 -> (forall {A} {κ : FalseH A ∈ Δ} {S : Sort A}
     -> ConH S ∈ hyps Ψ κ -> Ξ ⊨ ρ(κ) is RCon S)
 -> Ξ ⊨ sub(ρ) is RSub Ψ

tpexp : forall {Γ A}
         {κ : FalseH A ∈∈ Γ} {V : Γ ⊢ True A}
         {Ξ : RLCtx Γ} {S : Sort A}
 -> ConH S ∈ hyps' Ξ κ -> Ξ ⊨ V is RVal S
 -> Ξ ⊨ κ ∘ V is RExp
```

Again, whether or not you are fluent in Agda, it is worthwhile to pay attention to the overall structure of these rules. If we ignore the curly-bracketed implicit conditions, they look very much like extrinsic typing rules for raw syntax—but the implicit premises make clear that this is an extrinsic semantics of *typed* terms.

Another thing these rules are not is a refinement typing *algorithm.* The judgment $\Xi \vDash \texttt{t is } \mathcal{J}$ is defined as a coinductive family, and in order to verify that a term is well-sorted, we have to build an explicit witness auxiliary to the term. This runs somewhat counter to the original philosophy of refinement types, as a lightweight extension of ML-style typing—what we would really like is a decision procedure for refinement typing. But in order to obtain one, it turns out we will have to move from an infinitary, coinductive syntax, to a finitary, inductive one.

3.8 The trouble runs wide and deep

There are two main hurdles to directly defining a decision procedure for refinement typing on this representation of focusing proofs. One is that some types (such as `nat`) have infinitely many patterns, so that some metafunctions are "infinitely wide". Even if such metafunctions are represented constructively (as they are in Agda), refinement checking is still undecidable:

Proposition 28. *Assuming metafunctions may be defined by primitive recursion, there exists a class of closed* `nat`*-continuations for which refinement typing is undecidable.*

Proof. For any Turing Machine M, we define the `nat`-continuation $K_M = \mathsf{con}(\varphi_M)$ by primitive recursion on `nat`-patterns, as

$$\varphi_M(n) = \begin{cases} \mho & \text{If } M \text{ halts within } n \text{ steps} \\ \Omega & \text{otherwise} \end{cases}$$

Then $\vDash K_M \Leftarrow \top$ if and only if M never halts. □

The second problem is that the coinductive interpretation of the focusing rules allows terms that are "infinitely deep". Again, in practice such terms will be represented by a finite system of corecursive definitions in the metalogic—this was very convenient for defining Id_κ and *id*, for example—but how can we refinement check such terms *within* the metalogic, without being able to reflect on their representation?

4. A shallow, dis-functional syntax

We now systematically construct an alternative syntax for focusing proofs by applying the well-known programming languages techniques of *pattern-compilation* and *defunctionalization.* These address the two orthogonal concerns of §3.8: pattern-compilation produces terms that are finitely wide albeit infinitely deep, while defunctionalization gives these infinitely deep terms a finite, cyclic representation. In proof-theoretic terms, this attempt to relate infinitary and finitary derivations is very similar in spirit to the work of Mints (1978) and Buchholz (1991). These particular transformations seem to be closely related to the *sequent calculus for infinite descent* and *cyclic proofs* of Brotherston and Simpson (2007).

4.1 Pattern-compilation: introducing value variables

If we want to be able to replace an infinite collection of patterns with a finite one, we must be able to end pattern-matching early, binding a value rather than a continuation. So we update the definition of contexts (Def. 1) by including hypotheses "A *true*" in addition to "A *false*", and call a particular hypothesis A *true* $\in \Delta$ a **value variable** x. As a notational convenience, we write $\hbar$ to stand for either form of hypothesis, and $\ell :: (\hbar \in \Delta)$ for either kind of variable.

We now replace every pattern-formation rule with an *axiom,* i.e., a rule with no premises. For example, products and sums are redefined as follows:

$$\frac{}{\cdot \Vdash 1\ \mathit{true}} \qquad \frac{}{A\ \mathit{true}, B\ \mathit{true} \Vdash A \otimes B\ \mathit{true}}$$

$$\frac{}{A\ \mathit{true} \Vdash A \oplus B\ \mathit{true}} \qquad \frac{}{B\ \mathit{true} \Vdash A \oplus B\ \mathit{true}}$$

These rules are labelled with the **shallow patterns** $()$, $(\_,\_)$, $\mathsf{inl}\ \_$, and $\mathsf{inr}\ \_$ (which bind zero, two, one, and one value variable(s) respectively). Likewise, and more significantly, we redefine `nat` with the following axioms:

$$\frac{}{\cdot \Vdash \texttt{nat}\ \mathit{true}} \qquad \frac{}{\texttt{nat}\ \mathit{true} \Vdash \texttt{nat}\ \mathit{true}}$$

Whereas before there were infinitely many `nat`-patterns, now there are only two: `z` and `s _`.

After this shallow redefinition of patterns, the four rules of focusing need no modification except by generalizing the rule for building substitutions in the obvious way:

$$\frac{\hbar \in \Delta \quad \longrightarrow \quad \Gamma \vdash \hbar}{\Gamma \vdash \Delta}$$

Of course, this is not enough: we need a way of *using* value variables. Since the only way to use a continuation variable was to pass it a value, we might consider the symmetric rule:

$$\frac{A\ \mathit{true} \in \Gamma \quad \Gamma \vdash A\ \mathit{false}}{\Gamma \vdash \#}$$

But this is insufficiently general—for example, we can't even derive the identity principle on truth hypotheses. The solution is to first generalize the judgment A *false* to allow case-analysis towards an arbitrary term:

$$\frac{\Delta \Vdash A\ \mathit{true} \quad \longrightarrow \quad \Gamma, \Delta \vdash J}{\Gamma \vdash A\ \mathit{implies}\ J}$$

and then state the value variable rule as follows:

$$\frac{A\ \mathit{true} \in \Gamma \quad \Gamma \vdash A\ \mathit{implies}\ J}{\Gamma \vdash J}$$

We annotate this rule $\mathsf{case}\, x\, \mathsf{of}\, K$.

With this updated definition of focusing proofs, it is easy to derive the identity principle for value variables: for a variable $x ::$ $(A\ \mathit{true} \in \Gamma)$, the value $Id_x :: (\Gamma \vdash A\ \mathit{true})$ is defined by

$$Id_x = \mathsf{case}\, x\, \mathsf{of}\, \mathsf{con}(p \mapsto p[id])$$

Note, though, that again a coinductive interpretation of proofs is essential, because now even the identity coercion for `nat`s will be infinitely deep. It is also easy to verify the the cut principles: the $\mathsf{case}\ \_\ \mathsf{of}\ \_$ construct simply introduces *permutative con-*

versions. For example, the cut $K \bullet \mathsf{case}\, x\, \mathsf{of}\, \mathsf{con}(\varphi)$ reduces to $\mathsf{case}\, x\, \mathsf{of}\, \mathsf{con}(p \mapsto K \bullet \varphi(p))$.

The changes to the Agda embedding of §2.5 are minor, and we omit them for lack of space.

4.2 Defunctionalization

From a proof-theoretic point of view, defunctionalization amounts to picking a finite basis of combinators for building proofs, and in particular for building metafunctions. While this was already implicit when the higher-order rules were interpreted in a particular constructive setting (such as Agda), defunctionalization makes the commitment explicit. Moreover, the key property it gives us is the ability to define syntactic equality of metafunctions, using it as a decidable, conservative approximation to extensional equality.

Another very important consequence of defunctionalization is that we must *internalize* the cut principles as combinators, because we cannot always represent a program with a finite, cut-free derivation. Concretely, from a programmer's perspective, this just corresponds to the ordinary fact that sometimes we need to write auxiliary functions.

There are a few more subtleties to defunctionalization in our setting, although these become more or less obvious by "following the arrows" in Agda. For example, since there are two different kinds of metafunctions (φs and ρs) with different metatypes, we need two separate "apply" functions: `match` takes an identifier $\overline{\varphi}$ and a pattern p and returns the term corresponding to $\varphi(p)$, while `lookup` takes an identifier $\overline{\rho}$ and a variable ℓ and returns the corresponding value/continuation $\rho(\ell)$. The reader can consult the Agda encoding for an explanation of more such subtleties.

5. Decidable refinement types

After pruning the infinite width of terms by introducing value hypotheses, and taming their infinite depth by defunctionalization, we can define a decision procedure for refinement typing. Again we describe this algorithmic type system in two stages.

5.1 Refinement typing of pattern-compiled proofs

The pattern-inversion operator is revised to operate on shallow patterns just as one would expect, for example with

$$[\![(x, y) : S \otimes T]\!] = \{(x : S, y : T)\} \quad [\![\mathsf{s}\, x : \mathtt{even}]\!] = \{(x : \mathtt{odd})\}$$

The inversion rules for unions and intersections remain unchanged. The refinement typing rules require more subtle revision, however. It helps to understand why we can't directly use the value-typing rule based on deep patterns:

$$\frac{\Psi \in [\![p : S]\!] \quad \Xi \vDash \sigma : \Psi}{\Xi \vDash p[\sigma] : S}$$

Suppose, concretely, that we want to check the value $\mathsf{s}\, x[x \mapsto \mathit{Id}_y]$ (syntactic salt for "$\mathsf{s}\, y$") at sort $\mathtt{even} \cup \mathtt{odd}$. Pattern-inversion results in two possible refinement contexts: $(x : \mathtt{even})$ and $(x : \mathtt{odd})$. But we cannot necessarily verify that Id_y is either `even` or `odd`, because Ξ might only tell us that, say, $y : \mathtt{odd} \cup \mathtt{even}$. The problem is that the rule requires a non-invertible choice, and we cannot make a good one.

The solution is to eliminate the need for such choices by generalizing the refinement typing judgments to take a finite list of refinements, interpreted as follows:

Judgment	Interpretation
$\Xi \vDash V : \overrightarrow{S}$	V checks against a union of sorts
$\Xi \vDash K : \overrightarrow{S} > \mathcal{J}$	K takes the intersection of sorts to $\mathcal{J}$
$\Xi \vDash \sigma : \overrightarrow{\Psi}$	σ satisfies the union of refinement contexts
$\Xi \vDash E : \#$	E is well-sorted (as before)

The rule for value-typing now makes no premature choices:

$$\frac{\Xi \vDash \sigma : [\![p : S_1]\!], \ldots, [\![p : S_n]\!]}{\Xi \vDash p[\sigma] : S_1, \ldots, S_n}$$

Note this is equivalent to

$$\frac{\Xi \vDash \sigma : [\![p : S_1 \cup \cdots \cup S_n]\!]}{\Xi \vDash p[\sigma] : S_1, \ldots, S_n}$$

but we prefer the former presentation, since it does not mention any sort constructors. Likewise, we revise the rules for using variables, so that they do not require any choices:

$$\frac{\Xi \vDash K : \Xi(x) > \mathcal{J}}{\Xi \vDash \mathsf{case}\, x\, \mathsf{of}\, K : \mathcal{J}} \qquad \frac{\Xi \vDash V : \Xi(\kappa)}{\Xi \vDash \kappa\, V : \#}$$

The rule for typing continuations is functionally equivalent to

$$\frac{\Psi \in [\![p : S_1 \cap \cdots \cap S_n]\!] \quad \longrightarrow \quad \Xi, \Psi \vDash \varphi(p) : \mathcal{J}}{\Xi \vDash \mathsf{con}(\varphi) : S_1, \ldots, S_n > \mathcal{J}}$$

but we prefer the following presentation, which exploits the distributivity of unions through intersections (here $\bigwedge$ is the n-ary form of the context meet operation):

$$\frac{\Psi_1 \in [\![p : S_1]\!], \ldots, \Psi_n \in [\![p : S_n]\!] \quad \longrightarrow \quad \Xi, \bigwedge_i \Psi_i \vDash \varphi(p) : \mathcal{J}}{\Xi \vDash \mathsf{con}(\varphi) : S_1, \ldots, S_n > \mathcal{J}}$$

Also note that since continuations are now defined using shallow pattern-matching, the implication can be expanded into finitely many premises. Finally, we check substitutions against a union of refinement contexts, using the somewhat funny-looking:

$$\frac{\begin{array}{c}\ell_1 : \mathfrak{h}_1 \in \Psi_1, \ldots, \ell_n : \mathfrak{h}_n \in \Psi_n \quad \longrightarrow \\ \Xi \vDash \rho(x) : \bigwedge_i (\ell_i : \mathfrak{h}_i)(x) \text{ or } (\ell_i : \mathfrak{h}_i = \kappa \Leftarrow S \text{ and } \Xi \vDash \rho(\kappa) \Leftarrow S)\end{array}}{\Xi \vDash \mathsf{sub}(\rho) : \Psi_1, \ldots, \Psi_n}$$

This rule quantifies over every way of selecting hypotheses $x : S$ or $\kappa \Leftarrow S$ from the different refinement contexts, and then requires we show that the substitution satisfies the union of hypotheses for *some* variable. For value variables x, this means we collect all of the relevant facts $x : S_1, \ldots, x : S_n$, and show $\rho(x) : S_1, \ldots, S_n$. For continuation variables κ, on the other hand, we must find a particular hypothesis $\ell_i : \mathfrak{h}_i = \kappa \Leftarrow S$ and show $\rho(\kappa) \Leftarrow S$.

The substitution-typing rule might be a bit difficult to digest. It roughly resembles rules in the literature for subtyping in the presence of unions and products (Hosoya et al. 2000; Vouillon 2006). We can also observe a loose duality with the rule for checking continuations, particularly that this rule exploits the distributivity of intersections through unions.

5.2 Coinductive typing and sort annotations

Once we have the appropriate definition of refinement typing based on shallow patterns, all that remains is to make the coinductive interpretation of the typing rules effective via defunctionalization. To decide whether $\Xi \vDash t : \mathcal{J}$, we execute the above rules in a bottom-up manner, while also maintaining a growing list of assumptions χ about metafunctions in t. For example, when checking a metafunction φ against $S_1, \ldots, S_n > \mathcal{J}$, we add $\varphi : S_1, \ldots, S_n > \mathcal{J}$ as an assumption to χ, available when checking each of the (finitely many) pattern branches $\varphi(p)$. Using these assumptions requires comparing defunctionalized metafunctions for equality—as already mentioned, we do this syntactically.

Refinement typing of *cut-free* programs is completely automatic, precisely because all of the rules in §5.1 respect the subformula property and have finitely many premises. An implementation of this refinement typing procedure can be found in the Agda code. For example, we can automatically verify the expected properties of the examples `plus` and `app0` from §2.5, such as that `app0`

accepts sort $\mathtt{dom}^*$ (but not $\mathtt{dom}_{\mathtt{even}}$), and that `plus` takes a pair of `even`s to an `even` and a pair of `odd`s to an `even`, etc. Since the identity coercions are cut-free, we also immediately obtain a decision procedure for subtyping, by the identity coercion interpretation. For example, we can automatically verify that $\overset{v}{\neg}\mathtt{even} \cap \overset{v}{\neg}\mathtt{odd} \leq \overset{v}{\neg}(\mathtt{even} \cup \mathtt{odd})$, and that $\overset{v}{\neg}(\mathtt{even} \cap \mathtt{odd}) \not\leq \overset{v}{\neg}\mathtt{even} \cup \overset{v}{\neg}\mathtt{odd}$.

On the other hand, as we explained in §4.2, not every program can be written in a cut-free way. For example, the most natural way to write `times` is by cutting with `plus`. The refinement typing rules for cuts do not have the subformula property:

$$\frac{\Xi \vDash V : S \quad \Xi \vDash K : S > J}{\Xi \vDash K \bullet V : \mathcal{J}} \qquad \frac{\Xi, \Psi \vDash t : \mathcal{J} \quad \Xi \vDash \sigma : \Psi}{\Xi \vDash t[\sigma] : \mathcal{J}}$$

With sort annotations on these cuts, however, we can retain decidability. The practical issues of programming with annotations are outside the scope of this paper—for one, they must be *contextual annotations* (Dunfield and Pfenning 2004). The reader can consult the Agda code for a primitive implementation of contextual annotations, which, for example, allows us to verify programs such as `times` by placing the necessary annotations on `plus`.

6. Conclusions

We have derived a refinement type system with intersection and union types, that is inherently safe in the presence of effects. This type system is similar to earlier ones targeted at effectful call-by-value languages, particularly by Dunfield and Pfenning (2004), although we derive a stronger (while still sound) subtyping relationship. However, our aim here was not to exhibit a particular artifact, but rather to introduce a general, proof-theoretic methodology for designing type systems robust to effects, based on principles of duality.

We have moved systematically from a completely canonical, infinitary sequent calculus with an intrinsic proofs-as-programs interpretation, to an extrinsic refinement type system for these programs. Although this type system was undecidable, we could still use it to explore important properties such as subtyping, via the identity coercion and no-counterexamples interpretations. Moreover, we took the infinitary system as a starting point, from which we obtained a decidable system through well-known program transformation techniques. The only element that seemed to require a human oracle were the rules for shallow-pattern-based refinement typing in §5.1, but there too, perhaps further study will reveal a more systematic connection to the original rules.

Acknowledgments

Many thanks to Frank Pfenning, Dan Licata, Bob Harper, and Jeremy Avigad for valuable discussions, as well as to the anonymous PLPV referees, and the anonymous referees of an older version of this paper.

References

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

Franco Barbanera, Mariangiola Dezani-Ciancaglini, and Ugo De'Liguoro. Intersection and union types: syntax and semantics. *Information and Compution*, 119(2):202–230, 1995.

Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

James Brotherston and Alex Simpson. Complete sequent calculi for induction and infinite descent. In *IEEE Symposium on Logic in Computer Science*, pages 51–60, July 2007.

Wilfried Buchholz. Notation systems for infinitary derivations. *Archive for Mathematical Logic*, 30:277–296, 1991.

Pierre-Louis Curien and Hugo Herbelin. The duality of computation. In *ACM SIGPLAN International Conference on Functional Programming*, pages 233–243. 2000.

Olivier Danvy. On evaluation contexts, continuations, and the rest of the computation. In *CW'04: Proceedings of the Fourth ACM-SIGPLAN Continuation Workshop*, 2004.

Rowan Davies and Frank Pfenning. Intersection types and computational effects. In *ACM SIGPLAN International Conference on Functional Programming*, pages 198–208, 2000.

Joshua Dunfield and Frank Pfenning. Tridirectional typechecking. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 281–292, January 2004.

Andrzej Filinski. Declarative continuations and categorical duality. Master's thesis, University of Copenhagen, 1989. Computer Science Department.

Tim Freeman and Frank Pfenning. Refinement types for ML. In *Programming Language Design and Implementation*, pages 268–277, 1991.

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

Bob Harper and Mark Lillibridge. ML with callcc is unsound, 1991. Post to TYPES mailing list, July 8, 1991, archived at `http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html`.

Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. In *ACM SIGPLAN International Conference on Functional Programming*, pages 11–22, 2000.

Imre Lakatos. *Proofs and Refutations*. Cambridge University Press, New York, NY, 1976.

Søren B. Lassen and Paul Blain Levy. Typed normal form bisimulation. In Jacques Duparc and Thomas A. Henzinger, editors, *CSL*, volume 4646 of *Lecture Notes in Computer Science*, pages 283–297. Springer, 2007.

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

Paul-André Melliès and Jérôme Vouillon. Recursive polymorphic types and parametricity in an operational framework. In *IEEE Symposium on Logic in Computer Science*, pages 82–91, 2005.

Grigori E. Mints. Finite investigations of transfinite derivations. *Journal of Soviet Mathematics*, 10:548–596, 1978. Appears in Grigori Mints, *Selected papers in Proof Theory.* Bibliopolis, 1992.

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

Frank Pfenning. Church and Curry: Combining intrinsic and extrinsic typing. *Studies in Logic and the Foundations of Mathematics*, 2008. Festschrift in Honor of Peter B. Andrews on His 70th Birthday.

John C. Reynolds. The meaning of types: From intrinsic to extrinsic semantics. Report RS-00-32, University of Aarhus, December 2000.

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

Jérôme Vouillon. Polymorphic regular tree types and patterns. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 2006.

Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

Noam Zeilberger. On the unity of duality. *Annals of Pure and Applied Logic*, 153(1), 2008a. Special issue on "Classical Logic and Computation".

Noam Zeilberger. Focusing and higher-order abstract syntax. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 359–369, January 2008b.