Normalization by realizability also evaluates

Pierre-Évariste Dagand, Gabriel Scherer

Normalization by realizability also evaluates

Pierre-Évariste Dagand[1] & Gabriel Scherer[2]

1: Gallium, INRIA Paris-Rocquencourt
pierre-evariste.dagand@inria.fr
2: Gallium, INRIA Paris-Rocquencourt
gabriel.scherer@inria.fr

Abstract

For those of us that generally live in the world of syntax, semantic proof techniques such as realizability or logical relations/parametricity sometimes feel like magic. Why do they work? At which point in the proof is "the real work" done?

Bernardy and Lasson [4] express realizability and parametricity models as syntactic models – but the abstraction/adequacy theorems are still explained as meta-level proofs. Hoping to better understand the proof technique, we look at those proofs as programs themselves. How does a normalization argument using realizability actually computes those normal forms?

This detective work is at an early stage and we propose a first attempt in a simple setting. Instead of arbitrary Pure Type Systems, we use the simply-typed lambda-calculus.

1. Introduction

Realizability, logical relations and parametricity are tools to study the meta-theory of syntactic notions of computation or logic; typically, typed lambda-calculi. Starting from a syntactic type system (or system of inference rules for a given logic), they assign to each type/formula a predicate, or relation, that captures a semantic property, such as normalization, consistency, or some canonicity properties. Proof using such techniques rely crucially on an *adequacy lemma*, or *fundamental lemma*, that asserts that any term accepted by the syntactic system of inference also verifies the semantic property corresponding to its type – it is accepted by the predicate, or is in the diagonal of the relation. They are many variants of these techniques, which have scaled to powerful logics [1] (e.g., the Calculus of Constructions, or predicative type theories with a countable universe hierarchy) and advanced type-system features [12] (second-order polymorphism, general references, equi-recursive types...).

However, to some they have a feel of *magic*: while the proofs are convincing, one is never quite sure why they really work. Our long-term goal is to better understand the *computational behavior* of these techniques.

Where should we start looking, if we are interested in all three of realizability, logical relations and parametricity? Our reasoning was the following.

First, we ruled out parametricity: it is arguably a specific form of logical relation. Besides, it is motivated by applications – such as characterizing polymorphism, or extracting invariants from specific types – whose computational interpretation is less clear than a proof of normalization.

Second, general logical relations seem harder to work with than realizability. They are binary (or n-ary) while realizability is unary, and previous work [4] suggests that logical relations or parametricity can be built from realizability by an iterative process. While the binary aspect of logical relations seem essential to prove normalization of dependently-typed languages with a typed equality [2] (conveniently

modeled by the logical relation itself) or to formulate representation invariance theorems [3], it does not come into play for normalization proof of simpler languages such as the simply-typed lambda-calculus.

Among the various approaches to realizability (such as intuitionistic realizability, à la Prawitz, or classical realizability, à la Krivine), we decided to use classical realizability, which has two sets of realizers for each type A playing a symmetric role, namely the "truth witnesses" $|A|$ (or "proofs") and the "falsity witnesses" $\|A\|$ (or "counter-proofs"). While this choice was mostly out of familiarity, we think its symmetry creates a good setting to study both *negative* and *positive* connectives.

Keeping in mind that this is work is of exploratory nature, we would summarize our early findings as follows:

- The computational content of the adequacy lemma is an evaluation function.
- The polarity of object types seems to determine the flow of values in the interpreter.
- The way we define the set of truth and falsity witnesses determines the evaluation order. As could be expected, the bi-orthogonal injects a set of (co)-values inside a set of unevaluated (co)-terms.
- This evaluation order can be exposed by *recovering* the machine transitions from the typing constraints that appear when we move from a simply-typed version of the adequacy lemma to a dependently-version capturing the full content of the theorem.

1.1. The silhouette of realizability

Classical realizability is a technique to reason about the behavior of (abstract) *machines* (or *processes*, *configurations*), $c \in \mathbb{M}$, built as pairs $\langle\, t \mid e \,\rangle$ of a *term* $t \in \mathbb{T}$ and a *co-term* $e \in \mathbb{E}$. Those sets can be instantiated in many ways; for example, by defining $\mathbb{T}$ to be the terms of a lambda-calculus, and suitably choosing the co-terms, one may obtain results about the various evaluation strategies for this calculus. In this subsection, we will keep the framework abstract; to study the simply-typed lambda-calculus we will instantiate it in several slightly different ways.

Machines compute by reducing to other machines, as represented by a relation $(\rightsquigarrow) \subseteq \mathbb{M} \times \mathbb{M}$ between machines. They are closed objects, which have no other interaction with the outside world. The property of machines we want to reason about is represented by a *pole* $\perp\!\!\!\perp \subseteq \mathbb{M}$, a set of machines exhibiting a specific, good or bad, behavior – typically "machines that reduce to a normal form without raising an error". To be a valid notion of pole, the subset $\perp\!\!\!\perp$ must be closed by anti-reduction: if $c \rightsquigarrow c'$ and $c' \in \perp\!\!\!\perp$, we must have $c \in \perp\!\!\!\perp$.

While machines are closed objects, a term needs a co-term to compute, and vice-versa. If $\mathcal{T}$ is a set of terms, we define its *orthogonal* as the set of co-terms that "compute well" against terms in $\mathcal{T}$ – in the sense that they end up in the pole. The orthogonal of a set of co-terms is defined by symmetry:

$$\mathcal{T}^\perp \triangleq \{e \in \mathbb{E} \mid \forall t \in \mathcal{T}, \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp\} \qquad\qquad \mathcal{E}^\perp \triangleq \{t \in \mathbb{T} \mid \forall e \in \mathcal{E}, \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp\}$$

Orthogonality plays on intuitions of linear algebra, but it also behaves in a way that is similar to intuitionistic negation: for any set S of either terms or co-terms we have $S \subset S^{\perp\perp}$ (just as $A \implies \neg\neg A$) and this inclusion is strict in general, except for sets that are themselves orthogonals: $S^{\perp\perp\perp} \subseteq S^\perp$. These properties are easily proved – consider it an exercise for the reader. If one considers the pole to define a notion of *observation*, the bi-orthogonals of a given (co)term are those that behave in the same way. Being "stable by bi-orthogonality", that is being equal to its bi-orthogonal, is thus an important property; the sets we manipulate are often defined as orthogonals – or bi-orthogonals – of some more atomic sets [11].

While orthogonality is a way to describe the interface of a term or co-term in a semantic way, it can be hard to exhibit specific properties of (co)-terms by direct inspection of their orthogonal. It is useful to have a more explicit language to describe the interface along which a term and a co-term may interact to form a machine. Given a syntax for *types* (or logical *properties*) $A, B \ldots$, the terms interacting along the interface A are called the "truth witnesses" (or "truth values", or "proofs") of A, and written $|A|$. The co-terms interacting along the interface A are called the "falsity witnesses" (or "falsity values", or "counter-proofs", or "refutations") of A, and written $\|A\|$.

From a programming point of view, it also makes sense to think of $|A|$ as a set of "producers" of A, and $\|A\|$ as "consumers" of A. For example, one may define the falsity values of the arrow type, $\|A \to B\|$, as the (bi-orthogonal of the) product $|A| \times \|B\|$: to consume a function $A \to B$, one shall produce an A and consume a B.

What is all this used for? Two categories of applications are the following:

- We can use realizability to prove semantic properties of a syntactic system of judgments (type system, logic). Typically, the pole $\perp\!\!\!\perp$ is defined according to some (simple?) semantic intuition ("normalizes to a valid machine"), while a syntactic system of inference $\Gamma \vdash t : A$ is given by rules that specify no intrinsic correctness property. If all well-typed terms $\varnothing \vdash t : A$ happen to also be truth witnesses in $|A|$, this may tell us that our syntactic judgment does what it is supposed to. For example, we will prove normalization of the simply-typed lambda-calculus by showing that the machines c built from well-typed terms are in the pole $\perp\!\!\!\perp$ of normalizing machines.
- We can extend the language of machines, terms and co-terms with new constructs and reduction rules, in order to give a computational meaning to interesting logical properties which had no witnesses in the base language. Typically, extending the base calculus with a suitably-defined `callcc` construct provides a truth witness for the double-negation elimination (and, thus, to the excluded middle). When possible, this gives us a deeper (and often surprising) understanding of logical axioms.

We are concerned, in this article, only with the first aspect: using realizability to prove interesting properties of (relatively simple) type systems that classify untyped calculi. If the typing judgment for terms is of the form $\Gamma \vdash t : A$, and the typing judgment for co-terms of the form $\Gamma \mid e : B \vdash$, a well-typed machine $c : \Gamma \vdash$ is the pair of a well-typed term, and a well-typed co-term, interacting along the same type A.

Finally, one should note that truth witnesses (and, respectively, falsity witnesses and the pole) contain *closed* terms, which have no free variables. To interpret an open term $\Gamma \vdash t : A$, one should first be passed a substitution ρ that maps the free variables of t to closed terms; it is compatible with the typing environment Γ (which, in the judgments of the simply-typed lambda-calculus for example, maps term variables to types) if for each mapping $x : B$ in Γ, the substitution maps x to a truth witness for A: $\rho(x) \in |A|$. We write $|\Gamma|$ the set of substitutions satisfying this property.

For a particular syntactic system of inference, the typical argument goes by proving an *adequacy lemma*, itself formed of three mutually recursive results of the following form:

- for any t and ρ such as $\Gamma \vdash t : A$ and $\rho \in |\Gamma|$, we have $t[\rho] \in |A|$
- for any e and ρ such as $\Gamma \mid e : A \vdash$ and $\rho \in |\Gamma|$, we have $e[\rho] \in \|A\|$
- for any c and ρ such as $c : \Gamma \vdash$ and $\rho \in |\Gamma|$, we have $c[\rho] \in \perp\!\!\!\perp$

Interestingly, this can usually be done without fixing a single pole $\perp\!\!\!\perp$: adequacy lemmas tend to work for any pole closed under anti-reduction. Specializing it to different poles can then give different meta-theoretic results (weak and strong normalization, some canonicity results, etc.); in this article, we start simple and mostly use poles of the form "normalizes to a valid value".

1.2. Witnesses, value witnesses, and adequacy

As previously mentioned, we prefer sets of (co)terms to be stable by bi-orthogonality. In particular, we shall always define truth and falsity witnesses such that $|A|^{\perp\perp} \cong |A|$ and $\|A\|^{\perp\perp} \cong \|A\|$, for any A. We write $S \cong T$ to say that the sets S and T are in bijection, and reserve the equality symbol to the definitional equality.

To verify this property, we define those sets as themselves orthogonal of some more atomic sets of witnesses, dubbed "value witnesses". We assume a classification of our syntactic types A into a category P of *positive* types, characterized by their "truth value witnesses" $|P|_V$, and a category N of *negative* types, characterized by their "falsity value witnesses" $\|N\|_V$. Truth and falsity witnesses for constructed types can then be defined by case distinction on the polarity:

$$\begin{array}{lcllcl} \|P\| & \triangleq & |P|_V^{\perp} & \qquad & |P| & \triangleq & |P|_V^{\perp\perp} \\ \|N\| & \triangleq & \|N\|_V^{\perp\perp} & & |N| & \triangleq & \|N\|_V^{\perp} \end{array}$$

These definitions will be used in all variants of realizability proofs in this article: only $\|N\|_V$ and $|P|_V$ need to be defined, the rest is built on top of that. For consistency, we will also define $\|P\|_V \triangleq \|P\|$ and $|N|_V \triangleq |N|$: positives have no specific "falsity value witnesses", they are just falsity witnesses, and conversely negatives have only truth witnesses. Still, extending $|A|_V$ and $\|A\|_V$ to any type A, regardless of its polarity, gives us the following useful properties (which hold definitionally):

$$\forall A,\ |A| = \|A\|_V^{\perp} \qquad\qquad \forall A,\ \|A\| = |A|_V^{\perp}$$

This is directly checked by case distinction on the polarity of A. For a negative type for example, we have $|N| = \|N\|_V^{\perp}$ by definition, but also $|N|_V = |N| = \|N\|_V^{\perp}$ and $\|N\| = (\|N\|_V^{\perp})^{\perp}$, thus $\|N\| = |N|_V^{\perp}$.

As we remarked before, for any set S we have $(S^{\perp})^{\perp\perp} \cong S^{\perp}$. The above properties thus imply that truth and falsity witnesses, defined as orthogonals, are stable by bi-orthogonality.

1.3. Our approach

To distill the computational content of this proof technique, we implement the adequacy lemma *as a program* in a rather standard type theory[1]. As such, the result of this program is a proof of a set-membership $t \in |A|$, whose computational status is unclear (and is not primitively supported in most type theories).

Our core idea is to redefine these types in a proof-relevant way: we will redefine the predicate $\_ \in |\_|$ as a type of "witnesses": the inhabitants of $t \in |A|$ are the different syntactic justifications that the program t indeed realizes A. Note that $\_ \in |\_|$ is here an atomic construct, we do not define truth witnesses independently from their membership predicate.

In particular, we can pick the following proof-relevant definition of $\_ \in \perp\!\!\!\perp$: informally,

$$c \in \perp\!\!\!\perp \quad \triangleq \quad \{c \rightsquigarrow c_1 \rightsquigarrow \ldots \rightsquigarrow c_n \mid c_n \in \mathbb{M}_N\}$$

where $\mathbb{M}_N$ is the set of "normal machines", machines that are not stuck on an error but cannot reduce further. With this definition, an adequacy lemma proving $c \in \perp\!\!\!\perp$ for a well-typed c is exactly a normalization program – the question is *how* it computes.

While the definition of truth and falsity witnesses membership needs to be instantiated in each specific calculus, we can already give the general proof-relevant presentation of orthogonality, defining the types $\_ \in |\_|^{\perp}$ and $\_ \in \|\_\|^{\perp}$ as dependent function types:

$$e \in |A|^{\perp} \triangleq \Pi(t : \mathbb{T}).\, t \in |A| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp \qquad\qquad t \in \|A\|^{\perp} \triangleq \Pi(e : \mathbb{E}).\, e \in \|A\| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp$$

[1] Martin-Löf type theory with one universe suffices for our development. The universe is required to define the type of truth and falsity witnesses by recursion on the syntax of object-language types.

Finally, the semantic correctness of an environment $\rho \in |\Gamma|$ is defined itself as a substitution that maps any term variable x bound to the type A in Γ to a proof of membership $\rho(x) \in |A|$.

1.4. Simplifying further

As our very first step in the next section, we will use a simpler notion of pole that is not as orthodox, but results in simpler programs. This is crucial to make sense of their computational behavior. We will simply define $c \in \perp\!\!\!\perp \triangleq \mathbb{M}_N$: a witness that c is well-behaved is not a reduction sequence to some value v, but only that value v. This is a weaker type, as it does not guarantee that the value we get in return of the adequacy program is indeed obtained from c (it may be any other value). At some point in the following section, the more informative definition of $\perp\!\!\!\perp$ will be reinstated, and we will show how the simple programs we have written can be enriched to keep track of this reduction sequence – incidentally demonstrating that they were indeed returning the correct value.

A pleasant side-effect of this simplification is that the set membership types are not dependent anymore: the definition of $c \in \perp\!\!\!\perp$ does not depend on c ; definitions of $t \in |A|$ and $e \in \|B\|$ do not mention t, e ; $\rho \in |\Gamma|$ does not mention ρ ; and orthogonality is defined with non-dependent types.

To make that simplification explicit, we rename those types $\mathcal{J}(\perp\!\!\!\perp)$, $\mathcal{J}(|A|)$, $\mathcal{J}(\|B\|)$ and $\mathcal{J}(|\Gamma|)$: they are *justifications* of membership of some term (the type system does not remember which one), co-term or machine to the respective set. The definition of orthogonality becomes:

$$\mathcal{J}(|A|^\perp) \triangleq \mathcal{J}(|A|) \to \mathcal{J}(\perp\!\!\!\perp) \qquad\qquad \mathcal{J}(\|A\|^\perp) \triangleq \mathcal{J}(\|A\|) \to \mathcal{J}(\perp\!\!\!\perp)$$

In particular, this allows us to study the lambda-calculus without having to explicitly define the set of co-terms $\mathbb{E}$ first – which would steal some of the suspense away by forcing us to decide upon an evaluation order in advance. The realizability program will very much depend on the structure of falsity witnesses $\mathcal{J}(\|A\|)$ which we will define carefully, but not on the syntax of co-terms themselves – at first. In fact, we will show that we can *deduce* the syntax of co-terms from the dependent typing requirements, starting from the programs of the simplified version – so they need not be fixed in advance.

2. Normalization by realizability for the simply-typed lambda-calculus

We give a first example of a realizability-based proof of weak normalization of the simply-typed lambda-calculus with arrows and products.

				A, B	$:=$		types
t, u	$:=$		terms		$\mid$	X, Y, Z	atoms
	$\mid$	x, y, z	variables		$\mid$	$P \quad \mid \quad N$	constructed types
	$\mid$	$\lambda x.\, t$	abstractions				
	$\mid$	$t\ u$	applications	N	$:=$		negative types
	$\mid$	(t, u)	pairs		$\mid$	$A \to B$	function type
	$\mid$	$\texttt{let}\ (x, y) = t\ \texttt{in}\ u$	pair eliminations	P	$:=$		positive types
					$\mid$	$A * B$	product type

As mentioned in Section 1.2, we split our types into *positives* and *negatives*: our arrows are negative and our products are positive[2]. While this distinction has a clear logical status (it refers to which of the

[2]In intuitionistic logic, there is little difference between the positive and the negative product, so a product is always suspect of being negative in hiding. We could (and we have in our pen-and-paper work) use the sum instead, which is

left or right introduction rules, in a sequent presentation, are non-invertible), we will simply postulate it and use it to define truth and falsity witnesses. The type system describing the correspondence between the untyped terms and our small language of types is the expected one:

$$\Gamma, x{:}A \vdash x : A$$

$$\frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A * B} \qquad \frac{\Gamma \vdash t : A * B \qquad \Gamma, x{:}A, y{:}B \vdash u : C}{\Gamma \vdash \texttt{let}\ (x, y) = t\ \texttt{in}\ u : C}$$

Those inductive definitions of syntactic terms, types and type derivations should be understood as being part of the type theory used to express the adequacy lemma as a program: this program will manipulate syntactic representations of terms t, types A, and well-formed derivations (dependently) typed by a judgment $\Gamma \vdash t : A$. We color those syntactic objects (and the meta-language variables used to denote them) in blue to make it easier to distinguish, for example, the λ-abstraction of the object language $\lambda x.\, t$ and of the meta-language $\lambda x.\, t$. We will never mix identifiers differing only by their color – you lose little if you don't see the colors, we first wrote the article without them.

2.1. A realization program

Let us start with the following definitions of $|A|_V$ and $\|A\|_V$ as set of terms and co-terms:

$$|A * B|_V \triangleq |A| \times |B| \qquad \|A \to B\|_V \triangleq |A| \times \|B\|$$

In the simplified setting (defining datatypes of justifications $\mathcal{J}(\_)$ instead of membership predicates $\_ \in \_$), it is easy to make these proof-relevant, simply by interpreting the Cartesian product as a type of pairs – pairs of the meta-language:

$$\mathcal{J}(|A * B|_V) \triangleq \mathcal{J}(|A|) * \mathcal{J}(|B|) \qquad \mathcal{J}(\|A \to B\|_V) \triangleq \mathcal{J}(|A|) * \mathcal{J}(\|B\|)$$

Note that the types $\mathcal{J}(|A|_V)$, $\mathcal{J}(\|A\|_V)$ are *not* to be understood as inductive types indexed by an object-language type A, for they would contain fatal recursive occurrences in negative positions; for example,

$$\begin{aligned}\mathcal{J}(\|N \to A\|_V) &= \mathcal{J}(|N|) * \mathcal{J}(\|A\|) \\ &= \mathcal{J}(\|N\|_V^{\perp}) * \mathcal{J}(\|A\|) \\ &= (\mathcal{J}(\|N\|_V) \to \mathcal{J}(\perp\!\!\!\perp)) * \mathcal{J}(\|A\|)\end{aligned}$$

Instead, one should interpret $\mathcal{J}(|A|_V)$ and $\mathcal{J}(\|A\|_V)$ as mutually recursive type-returning *functions* defined by induction over the syntactic structure of A. We justified this with a Coq formalization 3.5.

With this in place, we can now write our "adequacy lemma" as a program. The adequacy lemma corresponds to providing a function of the following type:

$$\texttt{rea} : \forall\ \{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ (\Gamma \vdash t : A)\ \to \rho \in |\Gamma| \to t[\rho] \in |A|$$

which can be simplified further using the non-dependent version:

$$\texttt{rea} : \forall\ \{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ \{\Gamma \vdash t : A\}\ \to \mathcal{J}(|\Gamma|) \to \mathcal{J}(|A|)$$

necessarily positive in single-conclusions presentations, but this adds a non-negligible syntactic overhead (the elimination form has two premises, etc.) where we rather want to be light and concise for comprehensibility. You need to trust that we really do treat them positively – and it will be self-evident in the definition of truth witnesses.

The brackets around some arguments mean we will omit those arguments when writing calls to the `rea` function, as they can be non-ambiguously inferred from the other arguments – at least in the fully-dependent version – or the ambient environment.

We will present the code case per case, and discuss why each is well-typed. We will respect a specific naming convention for arguments whose dependent type is an inhabitation property: an hypothesis of type $t \in |A|$ (or $\mathcal{J}(|A|)$ in the simplified version) will be named $\bar{t}$, $\bar{v}$ for the type $t \in |A|_V$, $\bar{e}$ for the type $e \in \|A\|$, $\bar{\pi}$ for the type $\pi \in \|A\|_V$, and finally $\bar{\rho}$ for the type $\rho \in |\Gamma|$. Names like x, t, u, A, B, Γ will be used to name syntactic term variables, terms, types, and contexts. Finally, to help the reader type-check the code, we will write M^S if the expression M has type $\mathcal{J}(S)$. For example, $\bar{t}^{|A|}$ can be interpreted as $\bar{t} : \mathcal{J}(|A|)$.

Variable

$$\texttt{rea} \quad x \quad \{\Gamma, x{:}A \vdash x : A\} \quad \bar{\rho}^{|\Gamma, x:A|} \quad \triangleq \quad \bar{\rho}(x)^{|A|}$$

By hypothesis, the binding $x{:}A$ is in Γ, and we have $\bar{\rho} : \mathcal{J}(|\Gamma|)$, so in particular $\bar{\rho}(x) : x \in |A|$ as expected.

λ-abstraction We have structure information on the return type $\mathcal{J}(|A \to B|)$, which unfolds by definition to a function type $\mathcal{J}(\|A \to B\|_V) \to \mathcal{J}(\perp\!\!\!\perp)$. It is thus natural to start with a λ-abstraction over $\mathcal{J}(\|A \to B\|_V)$, that is, taking a pair of a $\bar{u} : \mathcal{J}(|A|)$ and a $\bar{e} : \mathcal{J}(\|B\|)$ to return a $\mathcal{J}(\perp\!\!\!\perp)$.

$$\texttt{rea} \quad (\lambda x.\, t) \quad \left\{ \frac{\Gamma, x{:}A \vdash t : B}{\Gamma \vdash \lambda x.\, t : A \to B} \right\} \quad \bar{\rho}^{|\Gamma|} \quad \triangleq \quad \lambda(\bar{u}^{|A|}, \bar{e}^{\|B\|}).\, \langle\, \texttt{rea}\ t\ \bar{\rho}[x \mapsto \bar{u}]^{|\Gamma, x:A|} \mid \bar{e} \,\rangle_B$$

The recursive call on $t : B$ gives a value of type $\mathcal{J}(|B|)$; we combine it with a value of type $\mathcal{J}(\|B\|)$ to give a $\mathcal{J}(\perp\!\!\!\perp)$ by using the auxiliary *cut function*

$$\langle\, \_ \mid \_ \,\rangle_A : \forall\ \{t\, e\}\,,\ t \in |A| \to e \in \|A\| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp$$

defined by case distinction on the polarity of the type A:

$$\langle\, \bar{t} \mid \bar{e} \,\rangle_P \triangleq \bar{t}\ \bar{e} \qquad\qquad \langle\, \bar{t} \mid \bar{e} \,\rangle_N \triangleq \bar{e}\ \bar{t}$$

For example in the positive case with $\bar{t}^{|P|}$ and $\bar{e}^{\|P\|}$, we have $|P| = \|P\|^\perp$ so the application $\bar{t}\ \bar{e}$ is well-typed.

Pair construction

$$\texttt{rea} \quad (t, u) \quad \left\{ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash (t, u) : A * B} \right\} \quad \bar{\rho} \quad \triangleq \quad (\texttt{rea}\ t\ \bar{\rho}, \texttt{rea}\ u\ \bar{\rho})^{\perp\perp}$$

The pair of recursive calls has type $\mathcal{J}(|A|) * \mathcal{J}(|B|)$, that is, $\mathcal{J}(|A * B|_V)$. This is not the expected type of the `rea` function, which is $\mathcal{J}(|A * B|) = \mathcal{J}(|A * B|_V^{\perp\perp})$. We used an auxiliary definition $\_^{\perp\perp}$ that turns any expression of type $\mathcal{J}(S)$ into $\mathcal{J}(S^{\perp\perp})$ – it witnesses the set inclusion $S \subseteq S^{\perp\perp}$. In fact, we shall define two distinct functions with this syntax, one acting on truth justifications (term witnesses), the other on falsity justifications (co-term witnesses).

$$(\bar{v}^{|P|_V})^{\perp\perp} \triangleq \lambda \bar{e}^{\|P\|}.\, \bar{e}\ \bar{v} \qquad\qquad (\bar{\pi}^{\|N\|_V})^{\perp\perp} \triangleq \lambda \bar{t}^{|N|}.\, \bar{t}\ \bar{\pi}$$

Notice that we have only defined $\_^{\perp\perp}$ on positive terms and negative contexts. We do not need it on negative terms, because it is not the case that $|N|_V^{\perp\perp} = |N|$ (definitionally). Instead, we will extend $\_^{\perp\perp}$ into a function $(\_)_V$ that embeds any $|A|_V$ into $|A|$, and any $\|B\|_V$ into $\|B\|$.

$$(\bar{v}^{|P|_V})_V \triangleq \bar{v}^{\perp\perp} \qquad (\bar{t}^{|N|_V})_V \triangleq \bar{t} \qquad (\bar{e}^{\|P\|_V})_V \triangleq \bar{e} \qquad (\bar{\pi}^{\|N\|_V})_V \triangleq \bar{\pi}^{\perp\perp}$$

Application

$$\texttt{rea}\quad (t\ u)\quad \left\{\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\right\}\quad \bar{\rho} \triangleq \lambda \bar{\pi}^{\|B\|_V}.\texttt{rea}\ t\ \bar{\rho}\ (\texttt{rea}\ u\ \bar{\rho}, (\bar{\pi})_V)$$

Unlike the previous cases, we know nothing about the structure of the expected return type $\mathcal{J}(|B|)$, so it seems unclear at first how to proceed. In such situations, the trick is to use the property $|B| = \|B\|_V^{\perp}$: the expected type is a function type $\mathcal{J}(\|B\|_V) \to \mathcal{J}(\perp\!\!\!\perp)$.

The function parameter $\bar{\pi}$ at type $\mathcal{J}(\|B\|_V)$ is injected into $\mathcal{J}(\|B\|)$ by the $(\_)_V$ function defined previously, and then paired with a recursive call on u at type $\mathcal{J}(|A|)$ to build a $\mathcal{J}(\|A \to B\|_V)$.

This co-value justification can then be directly applied to the recursive call on t, of type $\mathcal{J}(|A \to B|)$, using the property that $|N| = \|N\|_V^{\perp}$ for negative types N, that is $\mathcal{J}(|N|) = \mathcal{J}(\|N\|_V) \to \mathcal{J}(\perp\!\!\!\perp)$.

Pair destruction

$$\texttt{rea}\ (\texttt{let}\ (x,y) = t\ \texttt{in}\ u)\ \left\{\frac{\Gamma \vdash t : A * B \qquad \Gamma, x{:}A, y{:}B \vdash u : C}{\Gamma \vdash (\texttt{let}\ (x,y) = t\ \texttt{in}\ u) : C}\right\}\ \bar{\rho} \triangleq$$

$$\lambda \bar{\pi}^{\|C\|_V}.\langle\, \texttt{rea}\ t\ \bar{\rho} \mid \lambda(\bar{x},\bar{y})^{|A*B|_V}.\ \texttt{rea}\ u\ \bar{\rho}[x \mapsto \bar{x}, y \mapsto \bar{y}]\ \bar{\pi}\,\rangle_{A*B}$$

Here we use one last trick: it is not in general possible to turn a witness in $|A|$ into a witness in $|A|_V$ (respectively, a witness in $\|B\|$ into a witness in $\|B\|_V$), but it is when the return type of the whole expression is $\mathcal{J}(\perp\!\!\!\perp)$: we can combine our $\bar{t} : \mathcal{J}(|A|)$ with an abstraction $\lambda x^{|A|_V}.\ldots$ (thus providing a name x at type $\mathcal{J}(|A|_V)$ for the rest of the expression) returning a $\mathcal{J}(\perp\!\!\!\perp)$, using the cut function: $\langle\, \bar{t} \mid \lambda x.\, M^{\perp\!\!\!\perp}\,\rangle_A : \mathcal{J}(\perp\!\!\!\perp)$.

Summing up

$$\begin{array}{lcl}\langle\, \_ \mid \_\,\rangle_A & : & \mathcal{J}(|A|) \to \mathcal{J}(\|A\|) \to \mathcal{J}(\perp\!\!\!\perp)\\ \langle\, \bar{t} \mid \bar{e}\,\rangle_P & \triangleq & \bar{t}\ \bar{e}\\ \langle\, \bar{t} \mid \bar{e}\,\rangle_N & \triangleq & \bar{e}\ \bar{t}\end{array}$$

$$\begin{array}{lcl}\overline{(\bar{v}^{|P|_V})}^{\perp\perp} & : & \mathcal{J}(|P|_V) \to \mathcal{J}(|P|)\\ \overline{(\bar{v}^{|P|_V})}^{\perp\perp} & \triangleq & \lambda \bar{e}^{\|P\|}.\,\bar{e}\ \bar{v}\end{array}\qquad \begin{array}{lcl}\overline{(\bar{\pi}^{\|N\|_V})}^{\perp\perp} & : & \mathcal{J}(\|N\|_V) \to \mathcal{J}(\|N\|_V)\\ \overline{(\bar{\pi}^{\|N\|_V})}^{\perp\perp} & \triangleq & \lambda \bar{t}^{|N|}.\,\bar{t}\ \bar{\pi}\end{array}$$

$$\begin{array}{lcl}(\_)_V & : & \mathcal{J}(|A|_V) \to \mathcal{J}(|A|)\\ (\bar{v}^{|P|_V})_V & \triangleq & \bar{v}^{\perp\perp}\\ (\bar{t}^{|N|_V})_V & \triangleq & \bar{t}\end{array}\qquad \begin{array}{lcl}(\_)_V & : & \mathcal{J}(\|A\|_V) \to \mathcal{J}(\|A\|)\\ (\bar{e}^{\|P\|_V})_V & \triangleq & \bar{e}\\ (\bar{\pi}^{\|N\|_V})_V & \triangleq & \bar{\pi}^{\perp\perp}\end{array}$$

$$\begin{array}{llcl}\texttt{rea} & x^A & \bar{\rho} & \triangleq & \bar{\rho}(x)\\ \texttt{rea} & (\lambda x^A.\,t^B) & \bar{\rho} & \triangleq & \lambda(\bar{u}^{|A|}, \bar{e}^{\|B\|}).\langle\, \texttt{rea}\ t\ \bar{\rho}[x \mapsto \bar{u}] \mid \bar{e}\,\rangle_B\\ \texttt{rea} & (t^{A\to B}\ u^A) & \bar{\rho} & \triangleq & \lambda \bar{\pi}^{\|B\|_V}.\texttt{rea}\ t\ \bar{\rho}\ (\texttt{rea}\ u\ \bar{\rho}, (\bar{\pi})_V)\\ \texttt{rea} & (t^A, u^B) & \bar{\rho} & \triangleq & (\texttt{rea}\ t\ \bar{\rho}, \texttt{rea}\ u\ \bar{\rho})^{\perp\perp}\\ \texttt{rea} & (\texttt{let}\ (x,y) = t^{A*B}\ \texttt{in}\ u^C) & \bar{\rho} & \triangleq & \lambda \bar{\pi}^{\|C\|_V}.\langle\, \texttt{rea}\ t\ \bar{\rho} \mid \lambda(\bar{x},\bar{y}).\ \texttt{rea}\ u\ \bar{\rho}[x \mapsto \bar{x}, y \mapsto \bar{y}]\ \bar{\pi}\,\rangle_{A*B}\end{array}$$

It should be clear at this point that the computational behavior of this proof is, as expected, a normalization function. The details of how it works, however, are rather unclear to the non-specialist, in part due to the relative complexity of the types involved: the call $\texttt{rea}\ t\ \bar{\rho}$ returns a function type, so it may not evaluate its argument immediately. We shall answer this question in the next sections, in

which we present a systematic study of the various design choices available in the realizability proof. Finally, the next section moves from the simplified, non-dependent types to the more informative dependent types; which requires specifying a grammar of co-terms, which have been absent so far.

3. The many ways to realize the lambda-calculus

We made two arbitrary choices when we decided to define $\|A \to B\|_V \triangleq |A| \times \|B\|$. There are four possibilities with the same structure:

(1) $|A| \times \|B\|$ **(2)** $|A| \times \|B\|_V$ **(3)** $|A|_V \times \|B\|$ **(4)** $|A|_V \times \|B\|_V$

In this section, we would like to study the four possibilities, and see that each of them gives a slightly different realization program – but they all work. To reduce the verbosity, we will omit the positive product types from these variants; it can be handled separately (and would also give rise to four different choices, but they are less interesting as the construction is symmetric).

In Subsection 3.2, we shall also move from our simplified non-dependent definitions to "the real thing", which forces us to explicit our language of co-terms. This reveals the evaluation order from the typing constraints corresponding to the stability of the (dependently typed) pole by anti-reduction: two variants correspond to call-by-name, two to call-by-value.

3.1. Four realizations of the arrow connective

(1) $\|A \to B\|_V \triangleq |A| \times \|B\|$

$$\begin{array}{llcl} \texttt{rea} & x^A & \bar\rho & \triangleq & \bar\rho(x) \\ \texttt{rea} & (\lambda x^A.\, t^B)^{A\to B} & \bar\rho & \triangleq & \lambda(\bar u^{|A|}, \bar e^{\|B\|}).\, \langle\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B \\ \texttt{rea} & (t^{A\to B}\ u^A) & \bar\rho & \triangleq & \lambda \bar\pi^{\|B\|_V}.\, \texttt{rea}\ t\ \bar\rho\ (\texttt{rea}\ u\ \bar\rho, (\bar\pi)_V) \end{array}$$

(2) $\|A \to B\|_V \triangleq |A| \times \|B\|_V$

$$\begin{array}{llcl} \texttt{rea} & x^A & \bar\rho & \triangleq & \bar\rho(x) \\ \texttt{rea} & (\lambda x^A.\, t^B)^{A\to B} & \bar\rho & \triangleq & \lambda(\bar u^{|A|}, \bar\pi^{\|B\|_V}).\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u]\ \bar\pi \\ \texttt{rea} & (t^{A\to B}\ u^A) & \bar\rho & \triangleq & \lambda \bar\pi^{\|B\|_V}.\, \texttt{rea}\ t\ \bar\rho\ (\texttt{rea}\ u\ \bar\rho, \bar\pi) \end{array}$$

We should remark that there would be a different yet natural way to write the λ-case, closer to the $|A| \times \|B\|$ version: $\lambda(\bar u^{|A|}, \bar\pi^{\|B\|_V}).\, \langle\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid (\bar\pi)_V \,\rangle_B$ The good news is that those two versions are equivalent: simple equational reasoning shows that $\langle\, \bar t \mid (\bar\pi)_V \,\rangle_A$ is equivalent to $\bar t\ \bar\pi$, and conversely $\langle\, (\bar v)_V \mid \bar e \,\rangle_A$ is equivalent to $\bar e\ \bar v$.

The fact that a result can be computed in several, equivalent ways is not specific to this case. This phenomenon occurs in the three other variants as well. For example, in the application case of variant **(1)** above, it is possible to start with $\lambda \bar e^{\|B\|}.\, \_$ instead of $\lambda \bar\pi^{\|B\|_V}.\, \_$, and then take the resulting term of type $\mathcal{J}(\|B\|) \to \mathcal{J}(\perp\!\!\!\perp)$, that is $\mathcal{J}(\|B\|^\perp)$, and coerce it into a $\mathcal{J}(|B|)$ – this is the equality for positive B, but not for negatives. Again, the resulting program is equivalent to the version we gave. We will not mention each such situation (but we checked that they preserve equivalence), and systematically use the presentation that results in the simpler code.

We do not claim to have explored the entire design space, but would be tempted to conjecture that there is a unique pure program (modulo $\beta\eta$-equivalence) inhabiting the dependent type of `rea`.

(3) $\|A \to B\|_V \triangleq |A|_V \times \|B\|$ A notable difference in this variant – and the following one – is that we can restrict the typing of our environment witnesses to store value witnesses: rather than $\bar{\rho} : \rho \in |\Gamma|$, we have $\bar{\rho} : \rho \in |\Gamma|_V$, that is, for each binding $x : A$ in Γ, we assume $\bar{\rho}(x) : \rho(x) \in |A|_V$. The function would be typable with a weaker argument $\bar{\rho} : \rho \in |\Gamma|$ (and would have an equivalent behavior when starting from the empty environment), but that would make the type less precise about the dynamics of evaluation.

$$
\begin{array}{llll}
\texttt{rea} & x^A & \bar{\rho}^{|\Gamma|_V} \triangleq & (\bar{\rho}(x))_V \\
\texttt{rea} & (\lambda x^A.\, t^B)^{A \to B} & \bar{\rho} \triangleq & \lambda(\bar{v}^{|A|_V}, \bar{e}^{\|B\|}).\, \langle\, \texttt{rea}\ t\ \bar{\rho}[x \mapsto \bar{v}] \mid \bar{e}\, \rangle_B \\
\texttt{rea} & (t^{A \to B}\ u^A) & \bar{\rho} \triangleq & \lambda\bar{\pi}^{\|B\|_V}.\, \langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u^{|A|_V}.\, \texttt{rea}\ t\ \bar{\rho}\ (\bar{v}_u, (\bar{\pi})_V)\, \rangle_A
\end{array}
$$

Had we kept $\bar{\rho} : \rho \in |\Gamma|$, we would have only $\bar{\rho}(x)$ in the variable case, but $\bar{\rho}[x \mapsto (\bar{v})_V]$ in the abstraction case, which is equivalent if we do not consider other connectives pushing in the environment.

It is interesting to compare the application case with the one of variant **(1)**:

$$\lambda\bar{\pi}^{\|B\|_V}.\, \texttt{rea}\ t\ \bar{\rho}\ (\texttt{rea}\ u\ \bar{\rho}, (\bar{\pi})_V)$$

The relation between variant **(1)** and variant **(3)** seems to be an inlining of $\texttt{rea}\ u\ \bar{\rho}$, yet the shapes of the terms hint at a distinction between call-by-name and call-by-value evaluation. For now, we will only remark that those two versions are equivalent whenever A is negative, as in that case the cut $\langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u. \ldots \rangle_A$ applies its left-hand-side as an argument to its right-hand-side, giving exactly the definition of **(1)** after β-reduction.

(4) $\|A \to B\|_V \triangleq |A|_V \times \|B\|_V$

$$
\begin{array}{llll}
\texttt{rea} & x^A & \bar{\rho}^{|\Gamma|_V} \triangleq & (\bar{\rho}(x))_V \\
\texttt{rea} & (\lambda x^A.\, t^B)^{A \to B} & \bar{\rho} \triangleq & \lambda(\bar{v}^{|A|_V}, \bar{\pi}^{\|B\|_V}).\, \texttt{rea}\ t\ \bar{\rho}[x \mapsto \bar{v}]\ \bar{\pi} \\
\texttt{rea} & (t^{A \to B}\ u^A) & \bar{\rho} \triangleq & \lambda\bar{\pi}^{\|B\|_V}.\, \langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u^{|A|_V}.\, \texttt{rea}\ t\ \bar{\rho}\ (\bar{v}_u, \bar{\pi})\, \rangle_A
\end{array}
$$

In this variant, like in the previous one **(3)**, it seems rather tempting to strengthen the return type of the $\texttt{rea}\ t^A\ \bar{\rho}$ program to return not a $t \in |A|$, but $t \in |A|_V$. In particular, this would allow to remove the cut $\langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u. \ldots \rangle_A$ from the application case, instead directly writing $(\texttt{rea}\ u\ \bar{\rho}, \bar{\pi})^{\|A \to B\|_V}$. This strengthening, which seemed to coincide with the strengthening of the environment substitution from $\rho \in |\Gamma|$ to $\rho \in |\Gamma|_V$, does not lead to a well-typed program. The problem is with the application case $t^{A \to B}\ u$: while we can return a $\mathcal{J}(|B|)$ by using the "trick" that $|B| = \|B\|_V^{\perp}$, there is no way to build a $\mathcal{J}(|B|_V)$ without any knowledge of B.

3.2. Co-Terms and stronger, un-simplified types

It is now time to un-do the simplification presented in Section 1.4, by moving back to dependent types using the following definitions and types:

$$c \in \perp\!\!\!\perp \triangleq \{c \rightsquigarrow c_1 \rightsquigarrow \ldots \rightsquigarrow c_n \mid c_n \in \mathbb{M}_N\}$$

$$e \in |A|^{\perp} \triangleq \Pi(t : \mathbb{T}).\, t \in |A| \to \langle\, t \mid e\, \rangle \in \perp\!\!\!\perp \qquad t \in \|A\|^{\perp} \triangleq \Pi(e : \mathbb{E}).\, e \in \|A\| \to \langle\, t \mid e\, \rangle \in \perp\!\!\!\perp$$

$$\texttt{rea} : \forall\ \{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ (\Gamma \vdash t : A)\ \to \rho \in |\Gamma| \to t[\rho] \in |A|$$

Instead of starting from a known abstract machine for the lambda-calculus and its type system,

we propose to reverse-engineer the required machine from the code we have written in the previous subsection. Indeed, the typing constraints of the dependent version force us to exhibit a reduction sequence, that is to define the necessary machine transitions.

Consider then the case of λ-abstraction in variant **(1)**:

$$\texttt{rea} \quad (\lambda x^A.\, t^B)^{A\to B} \quad \bar\rho \quad \triangleq \quad \lambda(\bar u, \bar e)^{\|A\to B\|_V}.\, \langle\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B$$

We transform this code to a case of the dependently-typed, by adding annotations (and otherwise leaving the code structure unchanged). We can first look at the types: we need to lift the definition $\mathcal{J}(\|A \to B\|_V) \triangleq \mathcal{J}(|A|) * \mathcal{J}(\|B\|)$ of variant **(1)**. As we started from a set-theoretic definition $\|A \to B\|_V \triangleq |A| \times \|B\|$, it is natural to define the co-terms that inhabit $\|A \to B\|_V$, that is, the co-term indexes of the type $\_ \in \|A \to B\|_V$, as pairs of a term in A and a co-term in B:

$$(u, e) \in \|A \to B\|_V \triangleq (u \in |A|) * (e \in \|B\|)$$

For this definition to make sense, we must specify that pairs (u, e) of a $u : \mathbb{T}$ and $e : \mathbb{E}$ form valid co-terms. Then, the definition means that those pairs are the only co-terms that satisfy the predicate $\_ \in \|A \to B\|_V$ (defined by recursion on the syntactic type in $\|\_\|_V$). If this looks strange (we would get three inductive definitions that are defined by mutual recursion on one of their term-level indices), it can also be represented by the following, more heavyweight encoding without inductives, just a recursive functions from syntactic types of the object language to dependent products in our meta-language, writing $a \equiv b$ for the propositional equality – this is what our mechanized formalization does:

$$e_0 \in \|A \to B\|_V \triangleq \Sigma(u : \mathbb{T}, e : \mathbb{E}).\, (e_0 \equiv (u, e)) * (u \in |A|) * (e \in \|B\|)$$

For presentation purposes, we will keep using the inductive-like definition, which results in more readable programs. For example, the case $\texttt{rea}\ (\lambda x.\, t)$ in variant **(1)** can now be rewritten into the following dependently-typed version:

$$\texttt{rea}\ \{\Gamma\}\ \ (\lambda x^A.\, t^B)^{A\to B}\ \ \{\rho\}\ \ (\bar\rho : \rho \in |\Gamma|)\ \ \triangleq$$

$$\lambda(u, e) : \mathbb{E}.\, \lambda((\bar u : u \in |A|, \bar e : e \in \|B\|) : (u, e) \in \|A \to B\|_V).\ \langle\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B^{\texttt{lam}}$$

(Under the reading of $\_ \in \|A \to B\|_V$ as an inductive datatype, the reason why we can abstract over a pair-in-the-object-language $\lambda((u, e) : \mathbb{E})$ instead of the more general $\lambda(e_0 : \mathbb{E})$ is that the next pattern-matching necessarily teaches us that e_0 must be equal to a pair, so any other case would be an absurd pattern: matching on a pair covers all possible cases.)

The cut function was defined from the start with a dependent type:

$$\langle\, \_ \mid \_ \,\rangle_A : \forall\, \{t\, e\}\,,\ t \in |A| \to e \in \|A\| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp$$

but this is not enough to make the whole term type-check. Indeed, this gives to the expression $\langle\, \texttt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B^{\texttt{lam}}$ the type $\langle\, t[\rho, x \mapsto u] \mid e \,\rangle \in \perp\!\!\!\perp$. But we just abstracted on $(u, e) \in \|A \to B\|_V$, in order to form a complete term of type $(\lambda x.\, t)[\rho] \in \|A \to B\|_V^{\perp}$, so the expected type is $\langle\, (\lambda x.\, t)[\rho] \mid (u, e) \,\rangle \in \perp\!\!\!\perp$. This explains the small $\_^{\texttt{lam}}$ annotation in the code above: we define this as an auxiliary function of type

$$\_^{\texttt{lam}} : \forall\, \{x\, t\, u\, e\}.\, \langle\, t[x \mapsto u] \mid e \,\rangle \in \perp\!\!\!\perp \to \langle\, \lambda x.\, t \mid (u, e) \,\rangle \in \perp\!\!\!\perp$$

While we could think of several ways of inhabiting this function, by far the most natural is to specify our yet-unspecified machine reduction $c \rightsquigarrow c'$ as containing at least the following family of

reductions: $\langle\, \lambda x.\, t \mid (u, e) \,\rangle \rightsquigarrow \langle\, t[x \mapsto u] \mid e \,\rangle$.

Similarly, the application case can be dependently typed as

$$\texttt{rea}\ \{\Gamma\}\ \ (t^{A\to B}\ u^{A})\ \bar{\rho}\ \triangleq\ \lambda\pi : \mathbb{E}.\, \lambda\bar{\pi} : \pi \in \|B\|_V.\, \langle\, \texttt{rea}\ t\ \bar{\rho} \mid (\texttt{rea}\ u\ \bar{\rho}, (\bar{\pi})_V) \,\rangle^{\texttt{app}}_{A\to B}$$

The function $(\_)_V$, which we had define at the type $\mathcal{J}(\|A\|_V) \to \mathcal{J}(\|A\|)$, can be given – without changing its implementation – the more precise type $\forall\, \{\pi\}.\, \pi \in \|A\|_V \to \pi \in \|A\|$. This means that $\langle\, \texttt{rea}\ t\ \bar{\rho} \mid (\texttt{rea}\ u\ \bar{\rho}, (\bar{\pi})_V) \,\rangle_{A\to B}$ has type $\langle\, t[\rho] \mid (u[\rho], \pi) \,\rangle \in \perp\!\!\!\perp$. For the whole term to be well-typed, the auxiliary function $\_^{\texttt{app}}$ needs to have the type $\forall\, \{t\, u\, \pi\}.\, \langle\, t \mid (u, \pi) \,\rangle \in \perp\!\!\!\perp \to \langle\, t\ u \mid \pi \,\rangle \in \perp\!\!\!\perp$, which is inhabited by adding the family of reductions $\langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, t \mid (u, \pi) \,\rangle$.

Experts will have long recognized the Krivine abstract machine for the call-by-name λ-calculus. We would like to point out that some of it was to be expected, and some of it is maybe more surprising. It is not a surprise that we recovered the same shape of co-terms as in the Krivine machine: we started from the exact same definition of realizers, and realizers precisely determine what the co-terms are – even though our dependent, proof-relevant types partly obscures this relation. We argue that it is, however, less automatic that the *transitions* themselves would be suggested to us by typing alone – it means they are, in some sense, solely determined by the type of the adequacy lemma. In particular, we recovered the fact that the application transition needs to be specified only for *linear co-terms* π – in our setting, those that belong not only to $\|N\|$, but also to $\|N\|_V$.

3.3. Machine reductions and their variants

We will not repeat the process for all four variants of the arrow connective, but directly jump to the conclusion of what syntax for (arrow-related) co-terms they suggest, and which reduction rules they require to be well-typed. We observe that the two first variants correspond to a call-by-name evaluation strategy, while the two latter correspond to call-by-value; and, in particular, directly suggest the introduction of a $\tilde{\mu}x.c$ construction in co-terms.

(1) $\|A \to B\|_V \triangleq |A| \times \|B\|$

$$e ::= (u, e) \mid \ldots \qquad \langle\, \lambda x.\, t \mid (u, e) \,\rangle \rightsquigarrow \langle\, t[x \mapsto u] \mid e \,\rangle \qquad \langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, t \mid (u, \pi) \,\rangle$$

(2) $\|A \to B\|_V \triangleq |A| \times \|B\|_V$

$$e ::= (u, \pi) \mid \ldots \qquad \langle\, \lambda x.\, t \mid (u, \pi) \,\rangle \rightsquigarrow \langle\, t[x \mapsto u] \mid \pi \,\rangle \qquad \langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, t \mid (u, \pi) \,\rangle$$

(3) $\|A \to B\|_V \triangleq |A|_V \times \|B\|$ The application case is interesting in this setting. Its simplified version is the following:

$$\texttt{rea}\quad (t^{A\to B}\ u^{A})\quad \bar{\rho}\quad \triangleq\quad \lambda\bar{\pi}^{\|B\|_V}.\, \langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u^{|A|_V}.\, \langle\, \texttt{rea}\ t\ \bar{\rho} \mid (\bar{v}_u, (\bar{\pi})_V) \,\rangle_{A\to B} \,\rangle_A$$

This can be enriched to the following dependent version:

$$\texttt{rea}\ \{\Gamma\}\ \ (t^{A\to B}\ u^{A})\ \ \{\rho\}\ \ \bar{\rho} : \rho \in |\Gamma|_V\ \ \triangleq$$

$$\lambda\pi : \mathbb{E}.\, \lambda\bar{\pi} : \pi \in \|B\|_V.\quad \langle\, \texttt{rea}\ u\ \bar{\rho} \mid \lambda(v_u : \mathbb{T}).\, \lambda(\bar{v}_u : v_u \in |A|_V).\ \langle\, \texttt{rea}\ t\ \bar{\rho} \mid (\bar{v}_u, (\bar{\pi})_V) \,\rangle^{\texttt{app-body}}_{A\to B} \,\rangle^{\texttt{app-arg}}_{A}$$

The annotation $\_^{\texttt{app-body}}$ corresponds to a reduction to the machine $\langle\, t[\rho] \mid (v_u, \pi) \,\rangle$. Its return type – which determines the machine which should reduce to this one – is constrained not by the expected return type of the `rea` function as in previous examples, but by the typing of the outer cut

function which expects a $u[\rho] \in |A|$ on the left, and thus on the right a $e \in \|A\|$ for some co-term e. By definition of $e \in \|A\|$, the return type of $\_^{\mathtt{app-body}}$ should thus be of the form $\langle\, v_u \mid e \,\rangle \in \perp\!\!\!\perp$.

What would be a term e which would respect the reduction "equation"[3] $\langle\, v_u \mid e \,\rangle \rightsquigarrow \langle\, t[\rho] \mid (v_u, \pi) \,\rangle$ allowing to define $\_^{\mathtt{app-body}}$? This forces us to introduce System L's $\tilde{\mu}$ binder [5], to define $e \triangleq \tilde{\mu} v_x. \langle\, t[\rho] \mid (v_u, \pi) \,\rangle$, along with the expected reduction: $\langle\, v \mid \tilde{\mu} x.\, c \,\rangle \rightsquigarrow c[x \mapsto v]$.

From here, the input and output type of $\_^{\mathtt{app-arg}}$ are fully determined, and require the following reduction: $\langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, u \mid \tilde{\mu} v_u. \langle\, t \mid (v_u, \pi) \,\rangle \,\rangle$. Summing up:

$$e ::= (v, e) \mid \tilde{\mu} x.\, c \mid \dots \qquad \langle\, \lambda x.\, t \mid (v, e) \,\rangle \rightsquigarrow \langle\, t[x \mapsto v] \mid e \,\rangle \qquad \langle\, v \mid \tilde{\mu} x.\, c \,\rangle \rightsquigarrow c[x \mapsto v]$$

$$\langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, u \mid \tilde{\mu} x. \langle\, t \mid (x, \pi) \,\rangle \,\rangle$$

(4) $\|A \to B\|_V \triangleq |A|_V \times \|B\|_V$

$$e ::= (v, \pi) \mid \tilde{\mu} x.\, c \mid \dots \qquad \langle\, \lambda x.\, t \mid (v, \pi) \,\rangle \rightsquigarrow \langle\, t[x \mapsto v] \mid \pi \,\rangle \qquad \langle\, v \mid \tilde{\mu} x.\, c \,\rangle \rightsquigarrow c[x \mapsto v]$$

$$\langle\, t\ u \mid \pi \,\rangle \rightsquigarrow \langle\, u \mid \tilde{\mu} x. \langle\, t \mid (x, \pi) \,\rangle \,\rangle$$

3.4. Preliminary conclusions

The code itself may not be clear enough to dissect its evaluation order, but the machine transitions, that were imposed to us by the dependent typing requirement, are deafeningly explicit. When interpreting the function type $\|A \to B\|_V$, requiring truth *value* witnesses for A gives us call-by-value transitions, while just requiring arbitrary witnesses gives us call-by-name transitions.

The significance of choosing $\|B\|$ or $\|B\|_V$ as a second component is much less clear. It appears that reduction of applications $t\ u$ always happen against a linear co-term (one in $\|B\|_V$); it is only the reduction of the λ-abstraction that is made more liberal to accept a $\|B\|$ rather than a $\|B\|_V$. In particular, notice that the versions with $\|B\|_V$ always apply the continuation $\pi \in \|B\|_V$ directly to the recursive call on $t \in |B|$ in the $\lambda x.\, t$ case, while the $e \in \|B\|$ versions use a cut, meaning that the polarity of B will determine whether the realization of t is given the continuation as parameter, or the other way around. We conjecture that this different would be significant if our source calculus had some form control effects.

Finally, an interesting thing to note is that we have explored the design space of the semantics of the arrow connective independently from other aspects of our language – its product type. Any of these choices for $\|A \to B\|_V$ may be combined with any choice for $|A * B|_V$ (basically, lazy or strict pairs; the Coq formalization has a variant with strict pairs) to form a complete proof[4]. We see this as yet another manifestation of the claimed "modularity" of realizability and logical-relation approaches, which allows to study connectives independently from each other – once a notion of computation powerful enough to support them all has been fixed.

3.5. Partially mechanized formalization

The best way to make sure a program is type-correct is to run a type-checker on it. We formalized a part of the content of this article in a proof assistant. A Coq development that covers the non-dependent part – all four variants – with the pole left abstract is available at
http://gallium.inria.fr/~scherer/research/norm_by_rea/html/Norm.html

[3] We stole the idea of seeing introduction of a new co-term constructions as the resolutions of some equations from Guillaume Munch-Maccagnoni's PhD thesis [9].

[4] With the exception that it is only possible to strengthen the type of environments from $|\Gamma|$ to $|\Gamma|_V$ if all computation rules performing substitution only substitute values; that is a global change.

4. Related work

4.1. Intuitionistic realizability

Paulo Oliva and Thomas Streicher [10] factor the use of classical realizability for a negative fragment of second-order logic as the composition of intuitionistic realizability after a CPS translation.

The simplicity of this decomposition may unfortunately not carry on positive types, in particular the positive sum. In recent work, Danko Ilik [7] proposes a modified version of intuitionistic realizability to work with intuitionistic sums, by embedding the CPS translation inside the definition of realizability. The resulting notion of realizability is in fact quite close to our classical realizability presentation, with a "strong forcing" relation (corresponding to our set of *value* witnesses), and a "forcing" relation defined by double-negation ($\sim$ bi-orthogonal) of the strong forcing. In particular, Danko Ilik remarks that by varying the interplay of these two notions (notably, requiring a strong forcing hypothesis in the semantics of function types), one can move from a "call-by-name" interpretation to a "call-by-value" setting, which seems to correspond to our findings – we were not aware of this remark until late in the work on this article.

4.2. Normalization by Evaluation

There are evidently strong links with normalization by evaluation (NbE). The previously cited work of Danko Ilik [7] constructs, as is now a folklore method, NbE as the composition of a soundness proof (embedding of the source calculus into a mathematical statement parametrized on concrete models) followed by a strong completeness proof (instantiation of the mathematical statement into the syntactic model to obtain a normal form). In personal communication, Hugo Herbelin presented orthogonality and NbE through Kripke models as two facets of the same construction. Orthogonality focuses on the (co)-terms realizing the mathematical statement, exhibiting an untyped reduction sequence. NbE focuses on the typing judgments; its statement guarantees that the resulting normal form is at the expected type, but does not exhibits any relation between the input and output term.

4.3. System L

It is not so surprising that the evaluation function exhibited in Section 2.1 embeds a CPS translation, given our observation that the definition of truth and value witnesses determines the evaluation order. Indeed, the latter fact implies that the evaluation order should remain independent from the evaluation order of the meta-language (the dependent λ-calculus used to implement adequacy), and this property is usually obtained by CPS or monadic translations.

System L is a direct-style calculus that also has this propriety of forcing us to be explicit about the evaluation order – while being nicer to work with than results of CPS-encoding. In particular, Guillaume Munch-Maccagnoni shows that it is a good calculus to study classical realizability [8]. Our different variants of truth/falsity witnesses correspond to targeting different subsets of System L, which also determine the evaluation order. The principle of giving control of evaluation order to the term or the co-term according to polarity is also found in Munch-Maccagnoni PhD thesis [9].

The computational content of adequacy for System L has been studied, on the small $\mu\tilde{\mu}$ fragment, by Hugo Herbelin in his habilitation thesis [6]. The reduction strategy is fixed to be call-by-name. We note the elegant regularity of the adequacy statements for classical logic, each of the three versions (configurations, terms and co-terms) taking an environment of truth witnesses (for term variables) and an environment of falsity witnesses (for co-term variables).

4.4. Realizability in PTS

Jean-Philippe Bernardy and Marc Lasson [4] have some of the most intriguing work on realizability and parametricity we know of. In many ways they go above and beyond this work: they capture realizability predicates not as an ad-hoc membership, but as a rich type in a pure type system (PTS) that is built on top of the source language of realizers – itself a PTS. They also establish a deep connection between realizability and parametricity – we hope that parametricity would be amenable to a treatment resembling ours, but that is purely future work.

One thing we wanted to focus on was the *computational content* of realizability techniques, and this is not described in their work; adequacy is seen as a mapping from a well-typed term to a realizer inhabiting the realizability predicate. But it is a meta-level operation (not described as a *program*) described solely as an annotation process. We tried to see more in it – though of course, it is a composability property of denotational model constructions that they respect the input's structure and can be presented as trivial mappings (e.g., $[\![t\ u]\!] \triangleq \mathtt{app}([\![t]\!], [\![u]\!])$) given enough auxiliary functions.

Conclusion

At which point in a classical realizability proof is the "real work" done? We have seen that the computation content of the adequacy is a normalization function, that can be annotated to reconstruct the reduction sequence. Yet we have shown that there is very little leeway in proofs of adequacy: their computational content is determined by the *types* of the adequacy lemma and of truth and falsity witnesses. Finally, we have seen that *polarity* plays an important role in adequacy programs – even when they finally correspond to well-known untyped reduction strategies such as call-by-name or call-by-value. This is yet another argument to study the design space of type-directed or, more generally, polarity-directed reduction strategies.

Remerciements

Nous avons eu le plaisir de discuter avec Guillaume Munch-Maccagnoni, Jean-Philippe Bernardy, Hugo Herbelin et Marc Lasson; non contents d'avoir diffusé les idées au cœur de ce travail, ils se sont montrés disponibles et pédagogues. La relecture d'Adrien Guatto a apporté un œil extérieur indispensable, et les commentaires des rapporteurs anonymes et de Guillaume Munch-Maccagnoni nous ont permis d'améliorer encore grandement l'article.

References

[1] A. Abel. *Normalization by Evaluation: Dependent Types and Impredicativity*. PhD thesis, Ludwig-Maximilians-Universität München, May 2013. Habilitation thesis.

[2] A. Abel and G. Scherer. On irrelevance and algorithmic equality in predicative type theory. *Logical Methods in Computer Science*, 8(1), 2012.

[3] R. Atkey, N. Ghani, and P. Johann. A relationally parametric model of dependent type theory. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516, 2014.

[4] J. Bernardy and M. Lasson. Realizability and parametricity in pure type systems. In *Foundations of Software Science and Computational Structures - 14th International Conference, FOSSACS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 108–122, 2011.

[5] P. Curien and H. Herbelin. The duality of computation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243, 2000.

[6] H. Herbelin. C'est maintenant qu'on calcule, au cœur de la dualité. Thèse d'habilitation à diriger les recherches (French), 2011.

[7] D. Ilik. Continuation-passing style models complete for intuitionistic logic. *Ann. Pure Appl. Logic*, 164(6):651–662, 2013.

[8] G. Munch-Maccagnoni. Focalisation and Classical Realisability (version with appendices). In E. Grädel and R. Kahle, editors, *18th EACSL Annual Conference on Computer Science Logic - CSL 09*, volume 5771, pages 409–423, Coimbra, Portugal, 2009. Springer-Verlag.

[9] G. Munch-Maccagnoni. *Syntax and Models of a Non-Associative Composition of Programs and Proofs*. PhD thesis, Université Paris Diderot – Paris VII, 2013.

[10] P. Oliva and T. Streicher. On krivine's realizability interpretation of classical second-order arithmetic. *Fundam. Inform.*, 84(2):207–220, 2008.

[11] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 378–412, 2000.

[12] A. J. Turon, J. Thamsborg, A. Ahmed, L. Birkedal, and D. Dreyer. Logical relations for fine-grained concurrency. In *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*, pages 343–356, 2013.