Dependent Pearl: Normalization by realizability

PIERRE-ÉVARISTE DAGAND, Sorbonne Université, CNRS, Inria Paris, LIP6, France
LIONEL RIEG, VERIMAG UMR 5104, Université Grenoble Alpes, France, France
GABRIEL SCHERER, Inria Saclay, France

For those of us who generally live in the world of syntax, semantic proof techniques such as reducibility, realizability or logical relations seem somewhat magical despite – or perhaps due to – their seemingly unreasonable effectiveness. Why do they work? At which point in the proof is "the real work" done? Hoping to build a programming intuition of these proofs, we *implement* a normalization argument for the simply-typed λ-calculus with sums: instead of a proof, it is described as a program in a dependently-typed meta-language.

The semantic technique we set out to study is Krivine's *classical realizability*, which amounts to a proof-relevant presentation of *reducibility* arguments – unary logical relations. Reducibility assigns a predicate to each type, realizability assigns a set of *realizers*, which are abstract machines that extend λ-terms with a first-class notion of contexts. Normalization is a direct consequence of an *adequacy theorem* or "fundamental lemma", which states that any well-typed term translates to a realizer of its type. We show that the adequacy theorem, when written as a dependent program, corresponds to an evaluation procedure. In particular, a weak normalization proof precisely computes a series of reduction from the input term to a normal form. Interestingly, the choices that we make when we define the reducibility predicates – truth and falsity witnesses for each connective – determine the evaluation order of the proof, with each datatype constructor behaving in a lazy or strict fashion.

While most of the ideas in this presentation are folklore among specialists, our dependently-typed functional program provides an accessible presentation to a wider audience. In particular, our work provides a gentle introduction to abstract machine calculi which have recently been used as an effective research vehicle [Curien, Fiore, and Munch-Maccagnoni 2016; Downen, Johnson-Freyd, and Ariola 2015].

Additional Key Words and Phrases: classical realizability, dependent types, weak normalization, extraction

1 INTRODUCTION

Realizability, logical relations and parametricity are tools to study the meta-theory of syntactic notions of computation; typically, typed λ-calculi. Starting from a syntactic type system, they assign to each type/formula a predicate that captures a semantic property, such as normalization, consistency, or some canonicity properties. Proofs 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. In a realizability setting, one would prove that the term is accepted by the predicate. Similarly, the fundamental lemma of logical relations states that the term is accepted by the diagonal of the relation. There are many variants of these techniques, which have scaled to powerful logics [Abel 2013] (e.g., the Calculus of Constructions, or predicative type theories with a countable universe hierarchy) and advanced type-system features [Timany, Stefanesco, Krogh-Jespersen, and Birkedal 2018; Turon, Thamsborg, Ahmed, Birkedal, and Dreyer 2013] (second-order polymorphism, general references, equi-recursive types, local state…).

This paper sets out to explain 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 [Hermida, Reddy, and Robinson 2013]. 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 the previous work of Bernardy and Lasson [2011] suggests that logical relations and parametricity

can be built from realizability by an iterative process. While the binary aspect of logical relations is essential to formulate representation invariance theorems [Atkey, Ghani, and Johann 2014], it does not come into play for normalization proof of simpler languages such as the simply-typed λ-calculus. We are therefore left with realizability while keeping an eye towards normalization proofs – weak normalization rather than strong, for the benefit of simplicity. This is a well-worn path, starting with the reducibility method [Tait 1967], which we shall now recall and put in perspective with recent developments.

The notion of *polarity* plays a central role in the following retrospective. We split our types into either *positives* and *negatives*. This distinction has a clear logical status (it refers to which of the left or right introduction rules, in a sequent presentation, are non-invertible [Andreoli 1992; Zeilberger 2013]). This distinction is also reminiscent of call-by-push-value [Levy 2004] and can be summarized by the adage: "positive types are defined by their constructors whereas negatives types are defined by their destructors". The sum type, integers and inductive datatypes in general are positives whereas functions, records and coinductive types [Abel, Pientka, Thibodeau, and Setzer 2013] in general are negatives. ML programmers tend to favor the positive fragment whereas object-oriented programmers live almost exclusively in the negative fragment [Cook 2009]. In the absence of side-effects, we may sometimes playfully encode the latter using the former, through Church encodings for example, or we may gainfully defunctionalize the former to obtain the latter [Danvy 2006]. However, as noted before [Zeilberger 2009] and as we will see in the following, studying typed calculi through suitably polarized lenses allows us to distill 50 years of research into the following 3 pages.

On the negative fragment: reducibility. Normalization proofs for typed λ-calculi cannot proceed by direct induction on the term structure: to prove that a function application $t\ u$ is normalizing, a direct induction would only provide as hypotheses that the function t and its argument u are normalizing. But this tells us nothing of whether $t\ u$ itself normalizes; for example, the self-application $\omega \triangleq \lambda x.\, x\ x$ is normalizing, but applying it to itself gives a non-normalizable term $\omega\ \omega$ that reduces to itself.

A powerful fix to this issue, introduced by Tait [1967] and called reducibility, is to define a type-directed *reducibility* predicate: a term t of type A may or may not be "reducible at A". Reducibility is defined to imply normalization, but also provide stronger hypotheses to an inductive argument; in particular, we define "t is reducible at the function type $A \to B$" as the implication that if some u is "reducible at A", then the application $t\ u$ must be "reducible at B", which is exactly what we needed in the $t\ u$ case. One can then prove, by induction on typing derivations, that any well-typed term t of type A is normalizing at A.

There is more to this technique than a mere proof trick of strengthening an inductive hypothesis. In particular, we have transformed a *global* property of being normalizing into a *modular* property, reducibility, which describes the *interface* expected of terms at a given type. A global, whole-term property does not say much of what happens when a term is put in a wider context; the modular property specifies the possible interactions between the term and its context.

This view directly relates to more advanced fields of logic and type theory. It offers a semantic interpretation of types, where "behaving at the type A" is a property of how a (possibly untyped) term interacts with outside contexts; syntactic typing rules can be reconstructed as admissible proof principles [Harper 1992] that establish this behavioral property by syntactic inspection of the term. In logic, realizability is a general notion of building models where provable formulas are "realized" by computable functions, and it is through this lens that the constructive character of intuitionistic logics was historically explained. In programming language theory, it is common to

define not only a unary predicate in such a type-directed fashion, but also binary predicates, called logical relations, to capture the notion of program equivalence.

Negative interlude: a simpler proof of weak normalization. Remark that while reducibility is traditionally used to prove strong normalization – the fact that all possible reduction sequences (for all reduction strategies) are terminating – it can also be used to prove weak normalization – the fact that there exists one terminating reduction sequence. For the negative fragment of the simply-typed λ-calculus, there is a simpler proof argument for weak normalization, attributed to Alan Turing [Barendregt and Manzonetto 2013]. This argument does not proceed by induction on the typing derivation, but considers the sizes of the types appearing in redexes, and reduces one of the redexes with the largest types. The intuition is that when a redex $(\lambda x.\, t)\ u$ at type $A \to B$ is reduced to $u[t/x]$, it may create new redexes, but only at the smaller types A or B.

However, introducing sum types $A + B$ somewhat clouds this picture. The sum elimination construction is typed as follows

$$\frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_1 : A_1 \vdash u_1 : C \qquad \Gamma, x_2 : A_2 \vdash u_2 : C}{\Gamma \vdash \mathtt{match}\ t\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array} : C}$$

where C is unrelated in size to either A_1 or A_2. For example, consider the following term:

$$\mathtt{match} \left(\mathtt{match}\ \sigma_1\ v\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to \sigma_2\ (\sigma_2\ x_1) \\ \sigma_2\ x_2 \to \sigma_2\ (\sigma_1\ x_2) \end{array} \right) \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to 0 \\ \sigma_2\ x_2 \to 1 \end{array}$$

where v is a value of type A_1. The (only) available redex is the innermost case, at type $A_1 + A_2$. However, performing the reduction yields the term

$$\mathtt{match}\ \sigma_2\ (\sigma_2\ v)\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to 0 \\ \sigma_2\ x_2 \to 1 \end{array}$$

that introduces a redex of (larger!) type $A_3 + (A_2 + A_1)$. More generally, when the term t in the inference rule above is a sum injection $\sigma_i\ t'$; reducing this redex at type $A_1 + A_2$ into $u_i[t'/x_i]$ may create redexes at type A_i, but also at type C, which is unrelated in size. To do so, it is sufficient for a redex at C to appear when u_i starts with a constructor of type C and the reduced redex is under an elimination form for C, as exemplified by the above nested cases.

One can work around this issue by introducing commuting conversions. For instance, the nested cases above can be extruded as follows

$$\mathtt{match}\ \sigma_1\ v\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to \mathtt{match}\ \sigma_2\ (\sigma_2\ x_1)\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to 0 \\ \sigma_2\ x_2 \to 1 \end{array} \\ \sigma_2\ x_2 \to \mathtt{match}\ \sigma_2\ (\sigma_1\ x_2)\ \mathtt{with} \mid \begin{array}{l} \sigma_1\ x_1 \to 0 \\ \sigma_2\ x_2 \to 1 \end{array} \end{array}$$

whose effect is to enable the redexes of larger type – see for example Scherer [2016] for a systematic treatment of commuting conversions in a weakly normalizing setting. However, this approach adds bureaucracy and complexity.

Trying to naively extend the reducibility method to sum types hits a similar well-foundedness issue. A natural definition of reducibility at a sum type $A_1 + A_2$ would be that t is reducible at $A_1 + A_2$ if, for any type C and u_1, u_2 reducible at C, the elimination $(\mathtt{match}\ t\ \mathtt{with}\ \mid \sigma_1\ x \to u_1 \mid \sigma_2\ x \to u_2)$ is reducible at C. However, this definition is not well-founded. In the function case, we defined reducibility at $A \to B$ from reducibility at the smaller types A and B, but to define reducibility at type $A_1 + A_2$ we use the definition of reducibility at all types C, in particular at $A_1 + A_2$ itself or even larger types, such as $A_3 + (A_2 + A_1)$ in our example above.

On the positive fragment: bi-orthogonal closure. To obtain a reducibility proof accommodating sum types and side-effects, Lindley and Stark [2005] solved this well-foundedness problem. They use a bi-orthogonality closure, also called the top-top ($\top\top$) closure [Pitts and Stark 1998] following the notation of Pitts [2000], although ($\bot\bot$) would be a much more appropriate name and notation. Looking through the Curry-Howard isomorphism, we find that (linear) logicians had in fact pioneered this technique, Girard [1987] introducing bi-orthogonality to prove normalization of linear logic proof-nets. We follow in these footsteps (Section 3).

This technique proceeds as follows. We say that a program *context* $C\,[\Box]$ is "reducible at type A" if, for any *value* v of type A, the composed term $C\,[v]$ is normalizing. Then we define the terms reducible at a sum type $A + B$ to be the t such that, for any context $C\,[\Box]$ reducible at type $A + B$, the composed term $C\,[t]$ is normalizing.

The guiding intuition of our first attempt of a definition, and of the definition of reducibility at function types, was to express how "using" the term $t : A$ preserves normalization. This intuition still applies here; in particular, the set of contexts $C\,[\Box]$ includes the contexts of the form $\texttt{match}\ \Box\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2$, that perform a case analysis on their argument. However, notice that our definition does not require that $C\,[t]$ be reducible at its type, which would again result in an ill-founded definition, but only that it be normalizing: we do not require the modular property, only the global one.

Remark that adapting the naive definition by saying that t is reducible at $A_1 + A_2$ if

$$\texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x \to u \mid \sigma_2\ x \to u$$

is normalizing would give a well-founded definition, but one that is too weak, as the global property is not modular. For example, we would not be able to prove that a term of the form $(\texttt{match}\ t\ \texttt{with}\ \ldots)\ t'$ is normalizing by induction on its typing derivation. This is why we need to quantify on all contexts $C\,[\Box]$, such as $(\texttt{match}\ \Box\ \texttt{with}\ \ldots)\ t'$ in our example, rather than only on case eliminations.

There are really three distinct classes of objects in such a reducibility proof. The *term* and *context fragments* interact with each other, respecting a modular interface given by the reducibility predicate at their type. *Whole programs* are only required to be normalizing. These whole programs are formed by the interaction of a term fragment t and a context $C\,[\Box]$, both reducible at a type A.

Note that it is possible to adapt the standard reducibility arguments in presence of sums (t is reducible at $A_1 + A_2$ if it reduces to $\sigma_i\ t_i$ for t_i reducible at A_i) but this requires strengthening various (global) definitions to enforce stability under anti-reduction, which is a local property stating that if a predicate holds for $C\,[u_i[t_i/x_i]]$ then it must hold for $C\,[\texttt{match}\ \sigma_i\ t_i\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2]$ as well. This approach is thus less modular.

With this paper, we wish to turn these nuggets of wisdom into a rationalized process. We achieve this by expressing the problem in a setting familiar to functional programmers – dependent types – and by soliciting the shoulders of giant logicians (Section 2) to cast the problem is a mold so powerful as to make its solution simple and illuminating. This paper proposes the following 4-step program to enlightenment:

- We recall the standard framework of classical realizability for the simply-typed λ-calculus with sums while identifying the precise role of (co)-terms and (co)-values (Section 3). This distinction is absent from usual presentations of classical realizability [Miquel 2011; Rieg 2014] because of their bias toward negative types, whereas it plays a crucial role in our study of normalization proofs;

- We show that the computational content of the adequacy lemma is an evaluation function (Section 4). Along the way, we re-discover the fact that the polarity of object types determines the flow of values in the interpreter;
- We show that this evaluation order can be exposed by *recovering* the compilation to abstract machines from the typing constraints that appear when we move from a simply-typed version of the adequacy lemma to a dependently-typed version capturing the full content of the theorem (Section 5). Having carefully crafted our dependently-typed model, we use Coq to obtain a machine-checked proof of the adequacy lemma and Coq extraction mechanism to recover the underlying evaluation function in a systematic manner, thus validating our two constructions with one artefact;
- We illustrate how the construction of the set of truth and falsity witnesses mechanically sets the evaluation order of the resulting λ-calculus (Section 6). From this vantage point, we are lead to conclude that at the computational heart of a normalization proof lies an evaluator whose evaluation strategy is dictated by the flow of values in the model.

This research program is also a constructive one: throughout this paper, we demonstrate that, with some care, our argument[1] can be carried in the Coq proof assistant [The Coq Development Team 2018] and that the simply-typed normalization function from Section 4 can be extracted from the dependently-typed program of Section 5. We hope for this paper to rejoice logicians – by making fascinating techniques palatable to a larger audience of functional programmers – while quenching functional programmers' thirst for Curry-Howard phenomena.

2 BACKGROUND

This section recalls various notions this work relies on: realizability (giving a computational interpretation of the judgments of a logic or type system), classical realizability (realizability using abstract machines rather than λ-term), and the $\mu\tilde{\mu}$ abstract machine.

We color syntactic objects (and the meta-language variables used to denote them) in blue to make it easier to distinguish object-level constructs from meta-level constructs. This will become particularly useful starting from Section 4, where we will reify the meta-language into a program. We will never mix identifiers differing only by their color – you lose little if you do not see the colors.

2.1 The simply-typed λ-calculus

Figure 1 defines the usual simply-typed λ-calculus, with functions and sums. As we mentioned in the introduction, we distinguish *positive* types P and *negative* types N: our arrows are negative and our sums are positive. This distinction, which has a clear logical motivation, will matter when defining certain objects in our proofs.

2.2 Realizability interpretations

In mathematical logic, the name *realizability* describes a family of computational interpretations of logic, introduced by Kleene in 1945 to understand intuitionistic arithmetic.

Consider a logic with formulas A, and a set of syntactic inference rules for judgments of the usual form $\Gamma \vdash A$. We may wonder whether this logic is sound – cannot prove contradictions – and perhaps hope that it is constructive in some suitable sense. A realizability interpretation is given by a choice of syntax of *programs* p, and a set of *realizers* $|A|$ for each formula A of, such that (1) false formulas have no realizers and (2) closed, provable formulas $\vdash A$ have a realizer $p \in |A|$. This

[1] Available as supplementary material.

$$
\begin{array}{rcll}
t, u & := & & \text{terms} \\
& | & x, y, z & \text{variables} \\
& | & \lambda x.\, t & \text{abstractions} \\
& | & t\ u & \text{applications} \\
& | & \sigma_i\ t \qquad (i \in \{1, 2\}) & \text{injections} \\
& | & \texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2 & \text{case analysis}
\end{array}
$$

$$
\begin{array}{rcll}
A, B & := & & \text{types} \\
& | & X, Y & \text{type variables / atoms} \\
& | & P \quad | \quad N & \text{constructed types}
\end{array}
\qquad
\begin{array}{rcll}
N & := & & \text{negative types} \\
& | & A \to B & \text{function type} \\
P & := & & \text{positive types} \\
& | & A + B & \text{sum type}
\end{array}
$$

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

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

$$
\frac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2}
\qquad
\frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_i{:}A_i \vdash u_i : C}{\Gamma \vdash \texttt{match}\ t\ \texttt{with}\ \mid \sigma_1\ x_1 \to u_1 \mid \sigma_2\ x_2 \to u_2 : C}
$$

Fig. 1. Source language: simply-typed λ-calculus

yields a mathematical model that justifies the inference rules as well as provides a computational interpretation of the logic.

The realizability relation $p \in |A|$ – often written $p \Vdash A$ – plays a very different role from a typing judgment $\vdash p : A$. A type system ought to be modular and checking types ought to be decidable. Realizability can be defined in arbitrary ways, and tends to be wildly undecidable. Using powerful, untyped realizers allows us to realize more formulas than those provable in the initial logic and, in turn, any formula that we know how to realize can be added as an axiom without breaking soundness. In mathematical logic, realizability interpretations have been proposed for various axioms whose constructive character was not evident, such as Markov's principle, the excluded middle, the continuum hypothesis, and various flavors of the axiom of choice [Krivine 2014, 2009; Miquel 2011]. In computer science, realizability interpretations have been used to build models of powerful dependently-typed logics with set-theoretic connectives in the PRL family of proof assistants, or the soundness of advanced type systems for languages with side-effects [Brunel 2014; Lepigre 2016].

Classical realizability, presented next, is a sub-family of realizability interpretations whose realizers are *abstract machines*. It has been introduced by Krivine in the 90s to study the excluded-middle, using control operators to realize it and thus providing (constructive) models of classical logic [Krivine 2009] and Zermelo-Fraenkel set theory [Krivine 2014].

2.3 The Krivine machine: classical realizability in the negative fragment

As argued in the introduction, reasoning about λ-terms leads to considering not only the *terms* but also the *contexts* (with which the interaction occurs) and *whole programs* (which are just asked to

normalize). Abstract machines, such as the Krivine Abstract Machine [Krivine 2007][2] given just below, turn this conceptual distinction into explicit syntax: it includes terms t but also an explicit syntactic category of **e**valuation contexts e (context formers are $t \cdot e$, which pushes an applied argument t into the context e, and an end-of-stack symbol $\star$) as well as *machine configurations* $\langle\, t \mid e \,\rangle$, pairing a term and a context together to compute.

$$\langle\, t\ u \mid e \,\rangle \rightsquigarrow \langle\, t \mid u \cdot e \,\rangle \qquad (1)$$
$$\langle\, \lambda x.\, t \mid u \cdot e \,\rangle \rightsquigarrow \langle\, t[u/x] \mid e \,\rangle \qquad (2)$$

$$t ::= x \mid \lambda x.\, t \mid t\ u$$
$$e ::= \star \mid t \cdot e$$

Remark: for this simplified abstract machine, the terms t of the machine configuration are exactly the usual (untyped) λ-terms – with only function types, no sums. This will not be the case in general, so we are careful to distinguish the two syntactic categories; given a λ-term t, we write $\lfloor t \rfloor$ for its (identity) embedding as a machine term. Traditionally, logicians define type systems on top of realizers as way to systematically obtain realizers for large classes of formulas. In these systems, the terms *are* the realizers. However, recent work in classical realizability [Miquel 2018] demonstrates the value of decoupling both worlds, whereby one effectively compile terms to realizers.

This abstract machine simulates weak call-by-name reduction in the λ-calculus: the λ-term t reduces to a weak normal-form u if and only if the configuration $\langle\, \lfloor t \rfloor \mid \star \,\rangle$ reduces to $\langle\, \lfloor u \rfloor \mid \star \,\rangle$.

A configuration is a self-contained object, which has no other interaction with the outside world. Our realizability interpretation will be parametrized over the choice of a *pole* $\perp\!\!\!\perp$, a set of configurations exhibiting a specific, good or bad, behavior – for instance, "configurations that reduce to a normal form without raising an error".

Definition 2.1. A *realizability structure* [Krivine 2008] is a triple $(\mathbb{T}, \mathbb{E}, \perp\!\!\!\perp)$ where $\mathbb{T}$ is a set of machine terms, $\mathbb{E}$ a set of machine contexts, and $\perp\!\!\!\perp$ is a set of configurations $\langle\, t \mid e \,\rangle$ with $t \in \mathbb{T}$ and $e \in \mathbb{E}$ such that:

- $\mathbb{T}, \mathbb{E}$ are closed by context-formers:

$$\star \in \mathbb{E} \qquad\qquad t \in \mathbb{T} \ \wedge\ e \in \mathbb{E} \implies (t \cdot e) \in \mathbb{E}$$

- $\perp\!\!\!\perp$ is closed by anti-reduction:

$$\langle\, t' \mid e' \,\rangle \in \perp\!\!\!\perp \ \wedge\ \langle\, t \mid e \,\rangle \rightsquigarrow \langle\, t' \mid e' \,\rangle \implies \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp \qquad (3)$$

Unless explicitly noted, we will reason parametrically over any choice of realizability structure. We give a concrete instantiation at the end of the section, to study normalization.

While configurations are self-contained, a term needs a context to compute, and vice-versa. For $\mathcal{T} \subseteq \mathbb{T}$ an arbitrary set of terms, we define its *orthogonal* $\mathcal{T}^\perp$ as the set of contexts that "compute well" against terms in $\mathcal{T}$ – in the sense that they end up in the pole. The orthogonal of a set of contexts $\mathcal{E} \subseteq \mathbb{E}$ is defined by symmetry:

$$\mathcal{T}^\perp \triangleq \{e \in \mathbb{E} \mid \forall t \in \mathcal{T}, \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp\} \qquad (4) \qquad\qquad \mathcal{E}^\perp \triangleq \{t \in \mathbb{T} \mid \forall e \in \mathcal{E}, \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp\} \qquad (5)$$

Orthogonality behaves in a way that is similar to intuitionistic negation: for any set S of either terms or contexts we have $S \subseteq S^{\perp\perp}$ (just as $P \implies \neg\neg P$) and this inclusion is strict in general, except for sets that are themselves orthogonals: $S^{\perp\perp\perp} \subseteq S^\perp$ (just as $\neg\neg\neg P \implies \neg P$). If one understands the pole as defining a notion of *observation*, the bi-orthogonals of a given (co)-term are those that have the same observable behavior. In particular, we think of sets S that are "stable

[2] Although the reference paper is dated from 2007, as its author said, the KAM "was introduced twenty-five years ago". Its first appearance was in an unpublished but widely circulated note [Krivine 1985] available on the author's web page.

by bi-orthogonality" (S is isomorphic to $S^{\perp\perp}$) as observable predicates on terms or contexts: they define an extensional property that is not finer-grained than what we can observe through the pole. For example, taking the pole consisting of weakly normalizing configurations, the singleton set $S \triangleq \{\lambda x.\, x\}$ is not stable by bi-orthogonality: $\lambda x.\, (\lambda y.\, y)\ x$ does not belong to S whereas it is observationally indistinguishable from the identity function. Taking the bi-orthogonal closure of S allows us to encompass all those function that "behave like" the identity function from the standpoint of weak normalization.

We can now give a (classical) realizability interpretation of the types $A, B, \ldots$ of the simply-typed λ-calculus, parametrized over a choice of realizability structure $(\mathbb{T}, \mathbb{E}, \perp\!\!\!\perp)$. Rather than just defining sets of realizers $|A|$, we define a set of terms $|A|$ called the "truth witnesses" of A, and a set of contexts $\|A\|$ called the "falsity witnesses" of A – called this way because, from a logical point of view, $|A|$ contains "justifications" for A, while $\|A\|$ contains "refutations" for A. Their definitions imply that a term in $|A|$ and a context in $\|A\|$ can "interact according to the interface A".

$$\|A \to B\| \triangleq |A \to B|^{\perp} \tag{6}$$

$$|A \to B| \triangleq \|A \to B\|_V{}^{\perp} \tag{7}$$

$$\|A \to B\|_V \triangleq \{u \cdot e \mid u \in |A|, e \in \|B\|\} \tag{8}$$

Truth and falsity witnesses are both defined in terms of a set $\|A \to B\|_V$ of "falsity values", contexts of the form $t \cdot e$ where t is a truth witness for A, and e a falsity witness for B.

From a programming point of view, one can think of $|A|$ as a set of "producers" of A, and $\|A\|$ as a set of "consumers" of A. For example, the falsity values of the arrow type, $\|A \to B\|_V$, can be understood as "to consume a function $A \to B$, one shall produce an A and consume a B". Logically, the core argument in a refutation $A \to B$ is a justification of A paired with a refutation of B. Notice that $|A \to B|$ and $\|A \to B\|$ are both stable by bi-orthogonal, with $\|A \to B\| = |A \to A|^{\perp}$ by definition and $|A \to B| = \|A \to B\|^{\perp}$ from $S^{\perp\perp\perp} = S^{\perp}$. [3]

Realizability interpretations must ensure that provable formulas have a realizer. This is given by a Fundamental Theorem:

Theorem 2.2 (Fundamental Theorem). *If $\vdash t : A$ then $\lfloor t \rfloor \in |A|$*

To prove the fundamental theorem by induction on the typing derivation, we need to handle open terms. Typing environments Γ are realized by *substitutions* ρ mapping term variables to terms as expected: $\rho \in |\Gamma| \triangleq \forall (x : A) \in \Gamma, \quad \rho(x) \in |A|$. The fundamental theorem is then a direct corollary of the more precise Adequacy Lemma:

Lemma 2.3 (Adequacy). *If $\Gamma \vdash t : A$ then, for any $\rho \in |\Gamma|$, we have $\lfloor t \rfloor[\rho] \in |\Gamma|$.*

Proof. The proof proceeds by induction on the derivation.

Abstraction case.

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

Our goal is to prove that $(\lambda x.\, t)[\rho] \in |A \to B|$. Since $|A \to B| =^{(7)} \|A \to B\|_V{}^{\perp}$, we have to prove that for any context $u \cdot e$ with $u \in |A|$ and $e \in \|B\|$ we have $\langle\, (\lambda x.\, t)[\rho] \mid u \cdot e \,\rangle \in \perp\!\!\!\perp$. By anti-reduction (3) applied to (2), it suffices to show that $\langle\, t[\rho, u/x] \mid e \,\rangle \in \perp\!\!\!\perp$. The extended substitution

[3] We have not defined the truth and falsity witnesses of type variables (atomic types) X, Y above; we are parametric over any choice of subset of $\mathbb{T}$ and $\mathbb{E}$, as long as they are stable by bi-orthogonality.

$\rho, x/u$ is in $|\Gamma, x{:}A|$, so by induction hypothesis we have that $t[\rho, u/x] \in |B|$ – the body of the function behaves well at B. This suffices to prove $\langle\, t[\rho, u/x] \mid e \,\rangle \in \perp\!\!\!\perp$, given that e is in $\|B\|$, which is precisely the orthogonal of $|B|$.

Application case.

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}$$

We want to prove that $(t\ u)[\rho]$ is in $|B|$, that is: given any $e \in \|B\|$, we have $\langle\, (t\ u)[\rho] \mid e \,\rangle \in \perp\!\!\!\perp$. By anti-reduction (3) applied to (1), it suffices to prove $\langle\, t[\rho] \mid u[\rho] \cdot e \,\rangle \in \perp\!\!\!\perp$, which we obtain by induction hypothesis: $u[\rho]$ is in $|A|$, so $u[\rho] \cdot e$ is in $|A| \times \|B\| =^{(8)} \|A \to B\|_V$ which is included in $\|A \to B\|_V{}^{\perp\perp} =^{(7)} |A \to B|^{\perp} =^{(6)} \|A \to B\|$ and, since $t[\rho]$ is in $|A \to B|$, we can conclude that the configuration is in the pole. □

We can obtain various meta-theoretic results as consequences of the fundamental theorem, by varying the choice of the realizability structure $(\mathbb{T}, \mathbb{E}, \perp\!\!\!\perp)$. For example:

Corollary 2.4. *Closed terms in the simply-typed λ-calculus $\vdash t : A$ are weakly normalizing.*

Proof. We choose a particular realizability structure by defining $\perp\!\!\!\perp$ as the set of configurations that reach a normal form, $\mathbb{T}$ as the set of terms t such as $\langle\, t \mid \star \,\rangle$ is normalizing, and $\mathbb{E}$ as the set of contexts e such that $\langle\, x \mid e \,\rangle$ is normalizing. By definition of $\mathbb{T}$, we have that $\star$ is in $\|A\|$, that is $|A|^{\perp}$, for any A: indeed, for any term $t \in |A|$, we have $|A| \subset \mathbb{T}$ so, by definition, $\langle\, t \mid \star \,\rangle \in \perp\!\!\!\perp$.

By the Fundamental Theorem, $\vdash t : A$ implies $\lfloor t \rfloor \in |A|$. Given that $\star \in \|A\|$, we have $\langle\, \lfloor t \rfloor \mid \star \,\rangle \in \perp\!\!\!\perp$, that is, $\langle\, \lfloor t \rfloor \mid \star \,\rangle$ reduces to a (closed) normal form $\langle\, u \mid e \,\rangle$ – in fact u is exactly $\lfloor u \rfloor$. This can only be a normal form if t is a λ-abstraction and $e = \star$. We have deduced that $\langle\, \lfloor t \rfloor \mid \star \,\rangle \rightsquigarrow^* \langle\, \lfloor u \rfloor \mid \star \,\rangle$, so we know that $t \rightsquigarrow^* u$ in the λ-calculus, and u is a normal form. □

The Fundamental Theorem is also useful to establish various results based on the shape of normal forms for certain connectives; for example, we could easily prove that there is no closed term of the empty type. See for example Munch-Maccagnoni [2012]. In this paper, we will prove normalization of the simply-typed λ-calculus with sums by showing that the configurations m built from well-typed terms are in the pole $\perp\!\!\!\perp$ of normalizing configurations. Reading off the computational content from its proof, we shall discover a normalization function.

2.4 The $\mu\tilde{\mu}$ machine: classical realizability in the positive fragment too

We mentioned that realizers can be untyped, but they do not *need* to be untyped. In fact, there is a beautiful way to extend the type system for the simply-typed λ-calculus to terms of the abstract machine, with a judgment of the form $\Gamma \mid e : A \vdash B$ expressing that the context e consumes an input of type A to produce an output of type B.

$$\frac{\Gamma \vdash u : A \qquad \Gamma \mid e : B \to C}{\Gamma \mid u \cdot e : A \to B \vdash C} \qquad\qquad \frac{}{\Gamma \mid \star : A \vdash A}$$

This extension appeals to logicians: it corresponds to *sequent calculus*, a presentation of logic that has better cut-elimination properties than the usual *natural deduction*. Unfortunately, there is no direct way to extend the Krivine abstract machine with sums that would preserve this correspondence. Pragmatically, this means that reduction can get stuck unless we add commuting conversions.[4]

Fortunately, the work of Curien and Herbelin [2000] shows that the bias toward negativity can be avoided thanks to an abstract machine called $\mu\tilde{\mu}$. We define its syntax and dynamic semantics

[4] Which is why classical realizability is usually developed in System F, where datatypes are emulated by Church encoding.

$$m \in \mathbb{M} ::= \langle\, t \mid e \,\rangle \qquad \text{machine}$$

$t, u \in \mathbb{T} ::=$		terms	$e, f \in \mathbb{E} ::=$		co-terms
	$\mid x$	variable		$\mid \alpha$	co-variable
	$\mid \mu(x \cdot \alpha).\, m$	abstraction		$\mid t \cdot e$	function application
	$\mid \sigma_i\ t$	sum		$\mid \tilde{\mu}\,[x_1.\, m_1 \mid x_2.\, m_2]$	sum elimination
	$\mid \mu\alpha.\, m$	thunk		$\mid \tilde{\mu}x.\, m$	force

$$\frac{m_1' \rightsquigarrow m_2'}{m[m_1'] \rightsquigarrow m[m_2']}$$

$$\langle\, \mu\alpha.\, m \mid e \,\rangle \rightsquigarrow m[e/\alpha] \quad (9) \qquad \langle\, \mu(x \cdot \alpha).\, m \mid t \cdot e \,\rangle \rightsquigarrow m[t/x, e/\alpha] \quad (11)$$

$$\langle\, t \mid \tilde{\mu}x.\, m \,\rangle \rightsquigarrow m[t/x] \quad (10) \qquad \langle\, \sigma_i\ t \mid \tilde{\mu}\,[x_1.\, m_1 \mid x_2.\, m_2] \,\rangle \rightsquigarrow m_i[t/x_i] \quad (12)$$

Fig. 2. An untyped $\mu\tilde{\mu}$ abstract machine

(Figure 2).[5] Note that we are giving an untyped and unpolarized presentation of $\mu\tilde{\mu}$. It is known that this presentation is non-confluent; as we explained, the languages of realizers can have very wild computational behaviors. People working with $\mu\tilde{\mu}$ directly, instead of as a language of realizers, restrict the language to regain confluence by typing typing (enforcing normalization), evaluation strategies [Downen and Ariola 2018] or polarization [Munch-Maccagnoni 2013].

In the $\mu\tilde{\mu}$-calculus, some terms (or contexts) can capture the context (or term) set against them. The term $\mu\alpha.\, m$ binds its context to the name α and reduces to the machine configuration m. For example, in the Krivine machine, application $t\ u$ expects to be put against a context e, and this reduces to $\langle\, t \mid u \cdot e \,\rangle$. In $\mu\tilde{\mu}$, application $t\ u$ is not a primitive term-former. It can be encoded, following the above reduction principle, as $\mu\alpha.\, \langle\, t \mid u \cdot \alpha \,\rangle$. The symmetric construction exists for contexts $\tilde{\mu}x.\, m$ that capture the term set against them. Contexts are thus more powerful in the $\mu\tilde{\mu}$-machine than they are in the Krivine machine: they are in fact absolutely symmetrical to terms. We therefore call them *co-terms*, to conduce the idea that contexts too can drive reduction.

Finally, each connective is defined by a constructor and a pattern-matching form. The constructors for sums are the term $\sigma_1\ t$ and $\sigma_2\ t$, as in the λ-calculus. The pattern-matching form $\tilde{\mu}\,[x_1.\, m_1 \mid x_2.\, m_2]$ matches on the sum constructor, and reduces to one configuration or another depending on its value. For functions, the constructor is the context former $t \cdot e$ – negative types are defined by their *observations*, so the constructor builds a co-term. The pattern-matching form, $\mu(x \cdot \alpha).\, m$, matches on the application context and reduces to a configuration. In particular, $\lambda x.\, t$ is not a primitive, it can be encoded using the pattern-matching form, by defining it as $\mu(x \cdot \alpha).\, \langle\, t \mid \alpha \,\rangle$.

Figure 3 gives a translation scheme from plain λ-calculus terms to $\mu\tilde{\mu}$. This is not the only possible choice – as we will see in Section 6, varying the choice of translation gives different evaluation strategies.

The syntax of $\mu\tilde{\mu}$ may come as a bit of a shock to unfamiliar readers. Fortunately, most of our following sections can be understood without being familiar with $\mu\tilde{\mu}$, as the central object in our study is not the computational behavior of our realizers, but rather the computational meaning of the *proof* of the fundamental lemma, which we interpret using a (dependent) λ-calculus as the meta-language.

[5] We use a slight improvement of the original syntax, where λ is not a primitive anymore, due to Munch-Maccagnoni and Scherer [2015].

$$
\begin{array}{lcl}
\lfloor \lambda x.\, t \rfloor & \triangleq & \mu(x \cdot \alpha).\, \langle\, \lfloor t \rfloor \mid \alpha \,\rangle \\
\lfloor t\ u \rfloor & \triangleq & \mu\alpha.\, \langle\, \lfloor t \rfloor \mid \lfloor u \rfloor \cdot \alpha \,\rangle \\
\lfloor \sigma_i\ t \rfloor & \triangleq & \sigma_i\ \lfloor t \rfloor \\
\left\lfloor \texttt{match}\ t\ \texttt{with} \left| \begin{array}{l} \sigma_1\ x_1 \to u_1 \\ \sigma_2\ x_2 \to u_2 \end{array} \right| \right\rfloor & \triangleq & \mu\alpha. \left\langle \lfloor t \rfloor \mid \tilde{\mu} \left[\begin{array}{l} x_1.\, \langle\, \lfloor u_1 \rfloor \mid \alpha \,\rangle \\ x_2.\, \langle\, \lfloor u_2 \rfloor \mid \alpha \,\rangle \end{array} \right] \right\rangle
\end{array}
$$

Fig. 3. A compilation scheme from the λ-calculus to $\mu\tilde{\mu}$

3 INTERPRETING TYPES, PROGRAMMATICALLY

In this section, we set up the realizability framework in general terms, starting with a carefully engineered interpretation of types (Section 3.1) and a generic statement of the adequacy lemma (Section 3.2). We conclude with a specialization of the overall framework to a simple instance (Section 3.3) that focuses solely on the computational content of realizability, at the expense of its logical content.

3.1 Value witnesses & witnesses

In this paper, we assume a classification of our syntactic types A (Figure 1) 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" $\|N\|_V$. Sum types are inherently positive: they are defined by a choice of either a left or a right injection. Conversely, function types are inherently negative: they are defined by their behavior when applied to an argument in a given execution context. We postpone the actual definition of value witnesses until Section 4 for the special case of call-by-name functions and sums and Section 6 for a systematic study of the design space.

Truth and falsity witnesses for constructed types are defined by case distinction on the polarity:

$$\|P\| \triangleq |P|_V{}^{\perp} \tag{13}$$
$$\|N\| \triangleq \|N\|_V{}^{\perp\perp} \tag{14}$$
$$|P| \triangleq |P|_V{}^{\perp\perp} \tag{15}$$
$$|N| \triangleq \|N\|_V{}^{\perp} \tag{16}$$

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 also define

$$\|P\|_V \triangleq \|P\| \tag{17}$$
$$|N|_V \triangleq |N| \tag{18}$$

Indeed, 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):

$$\forall A,\ |A| = \|A\|_V{}^{\perp} \tag{19}$$
$$\forall A,\ \|A\| = |A|_V{}^{\perp} \tag{20}$$

This is directly checked by case distinction on the polarity of A. For a negative type, for example, we have $|N| =^{(16)} \|N\|_V{}^{\perp}$, but also $|N|_V =^{(18)} |N| =^{(16)} \|N\|_V{}^{\perp}$ and $\|N\| =^{(14)} (\|N\|_V{}^{\perp})^{\perp}$, thus $\|N\| = |N|_V{}^{\perp}$.

As we remarked before, for any set S we have $(S^{\perp})^{\perp\perp} \cong S^{\perp}$, where 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. As a result, we can check that $|A|^{\perp\perp} \cong |A|$ and $\|A\|^{\perp\perp} \cong \|A\|$, for any type A. Meaning that truth and falsity witnesses, defined as orthogonals, are indeed stable by bi-orthogonality.

3.2 Adequacy

To distill the computational content of realizability, we implement the adequacy lemma *as a program* in a rather standard type theory. 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.

Truth witnesses (and, respectively, falsity witnesses and the pole) contain *closed* programs, 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. Such substitution is compatible with the typing environment Γ if for each mapping $x : A$ in Γ, the substitution maps x to a truth witness for A: $\rho(x) \in |A|$. We use $|\Gamma|$ to denote the set of substitutions satisfying this property.

In this setting, the adequacy lemma amounts to a function of the following type:

$$\mathtt{rea} : \forall\{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ \{\Gamma \vdash t : A\} \to\ \rho \in |\Gamma|\ \to\ \lfloor t \rfloor[\rho] \in |A|$$

As such, the result of this program is a proof of a set-membership $\lfloor t \rfloor \in |A|$, 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" where the inhabitants of $\lfloor t \rfloor \in |A|$ are the different syntactic justifications that the $\mu\tilde{\mu}$ term $\lfloor t \rfloor$ indeed realizes A. We will respect a specific naming convention for arguments whose dependent type is an inhabitation property: an hypothesis of type $t \in |A|$ will be named $\bar{t}$, $\bar{v}$ for the type $v \in |A|_V$ of ***values***, $\bar{e}$ for the type $e \in \|A\|$ of co-terms, $\bar{\pi}$ for the type $\pi \in \|A\|_V$ of *linear co-terms*[6], and finally $\bar{\rho}$ for the type $\rho \in |\Gamma|$. Names like x, t and u, A, B, Γ will be used to name syntactic term variables, terms, types, and co-terms.

Interestingly, we did not have to fix a single pole $\perp\!\!\!\perp$ yet: the adequacy lemma actually works 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, specification problems, *etc.*). In this paper, we pick the following proof-relevant definition of $\_ \in \perp\!\!\!\perp$: informally,

$$m \in \perp\!\!\!\perp \quad \cong \quad \{m_n \in \mathbb{M}_N \mid m \rightsquigarrow m_1 \rightsquigarrow \ldots \rightsquigarrow m_n\} \tag{21}$$

where $\mathbb{M}_N$ is the set of "normal configurations", *i.e.*, configurations that are not stuck on an error but cannot reduce further. With this definition, an adequacy lemma proving $m \in \perp\!\!\!\perp$ for a well-typed m 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, treating the types $\_ \in |\_|^\perp$ and $\_ \in \|\_\|^\perp$ as dependent function types[7]:

$$e \in |A|^\perp \cong \forall\{t : \mathbb{T}\}.\, t \in |A| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp \qquad t \in \|A\|^\perp \cong \forall\{e : \mathbb{E}\}.\, e \in \|A\| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp$$

The orthogonality between truth and falsity witnesses is computationally incarnated by a *cut* function

$$\langle\, \_ \mid \_ \,\rangle_A : \forall\ \{t\, e\}\,,\ t \in |A| \to e \in \|A\| \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp \tag{22}$$

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

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

[6]Linear co-terms are to co-terms what values are to terms. Extensionally, this means that $\forall t$, $\langle\, t \mid \pi \,\rangle \not\rightsquigarrow$ whereas, intensionally, this amounts to having $\pi = t_1 \cdot \ldots \cdot \star$ (*i.e.*, context constructors), justifying the adjective "linear" in the sense that it contains a unique continuation $\star$. The notation π comes from the similarity between the French pronunciations of π and stack: π is pronounced [pi] whereas stack, ***pile*** in French, is pronounced [pil].

[7]Following common usage, we write $\forall\{x : A\}.\, B$ to declare an implicit argument of type A, meaning that we will omit those arguments when writing calls to function of such type, as they can be non-ambiguously inferred from the other arguments or the ambient environment. We write $\lambda\{a\}.\, t$ if we want to explicitly bind an implicit argument.

For example in the positive case with $\bar{t}^{|P|}$ and $\bar{e}^{\|P\|}$, we have $|P| =^{(15)} \|P\|^{\perp}$ so the application $\bar{t}\ \bar{e}$ is well-typed. A similar reasoning holds for a negative type N. Intuitively, the cut function reflects the intuition that interacting at a positive type gives control to the term (*i.e.*, reduction is strict) whereas interacting at a negative type gives control to the co-term (*i.e.*, reduction is lazy).

Similarly, the set inclusion $S \subseteq S^{\perp\perp}$ admits a computational interpretation in the form of an auxiliary definition $\_^{\perp\perp}$. For conciseness, we overload this syntax to define two (distinct) functions, one acting on truth witnesses, the other on falsity witnesses.

$$\begin{array}{ll} \_^{\perp\perp} : \forall\{P\ v\}.\ v \in |P|_V \to v \in |P| & \_^{\perp\perp} : \forall\{N\ \pi\}.\ \pi \in \|N\|_V \to \pi \in \|N\| \\ (\bar{v})^{\perp\perp} \triangleq \lambda\bar{e}^{\|P\|}.\ \bar{e}\ \bar{v} & (\bar{\pi})^{\perp\perp} \triangleq \lambda\bar{t}^{|N|}.\ \bar{t}\ \bar{\pi} \end{array}$$

Notice that we have only defined $\_^{\perp\perp}$ on positive terms and negative co-terms. We do not need it on negative terms, because we have that $\|P\|_V =^{(17)} \|P\|$ and $|N|_V =^{(18)} |N|$. Instead, just as we generalized $|\_|_V$ and $\|\_\|_V$ to be defined for any type, we further overload $\_^{\perp\perp}$ to transparently embed any $|A|_V$ into $|A|$, and any $\|B\|_V$ into $\|B\|$, irrespective of polarity.

$$(\bar{v}^{|P|_V})^{\perp\perp} \triangleq \bar{v}^{\perp\perp} \qquad (\bar{t}^{|N|_V})^{\perp\perp} \triangleq \bar{t} \qquad (\bar{e}^{\|P\|_V})^{\perp\perp} \triangleq \bar{e} \qquad (\bar{\pi}^{\|N\|_V})^{\perp\perp} \triangleq \bar{\pi}^{\perp\perp} \tag{23}$$

Once again, computational breadcrumbs can be gathered from this function definition: a value (resp., linear co-term) leaves it entirely to the opposing co-term (resp., term) to build the reduction witnesses, as indeed the value (resp., linear co-term) is already in normal form.

3.3 Simplifying further

In the next section, we will use a simpler notion of pole that is not as orthodox, but results in simpler programs. This pedagogical detour shall allow us to get to acquainted with the computational content of the adequacy lemma. In effect, we will simply define $m \in \perp\!\!\!\perp \triangleq \mathbb{M}_N$: a witness that m is well-behaved is not a reduction sequence to a normal configuration v, but only that configuration 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 m (it may be any other normal configuration). In Section 5, the more informative definition of $\perp\!\!\!\perp$ will be reinstated, and we will show how the simple programs we are going to write 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 $m \in \perp\!\!\!\perp$ does not depend on m ; 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 track which one), co-term or configuration to the respective set:

$$\mathcal{J}(\perp\!\!\!\perp) \triangleq \mathbb{M}_N \tag{21'}$$

$$\mathcal{J}(|A|^{\perp}) \triangleq \mathcal{J}(|A|) \to \mathcal{J}(\perp\!\!\!\perp) \quad (4') \qquad \mathcal{J}(\|A\|^{\perp}) \triangleq \mathcal{J}(\|A\|) \to \mathcal{J}(\perp\!\!\!\perp) \quad (5')$$

$$\mathcal{J}(\|P\|) \triangleq \mathcal{J}(|P|_V{}^{\perp}) \quad (13') \qquad \mathcal{J}(|P|) \triangleq \mathcal{J}(|P|_V{}^{\perp\perp}) \quad (15')$$

$$\mathcal{J}(\|N\|) \triangleq \mathcal{J}(\|N\|_V{}^{\perp\perp}) \quad (14') \qquad \mathcal{J}(|N|) \triangleq \mathcal{J}(\|N\|_V{}^{\perp}) \quad (16')$$

In particular, this allows us to study the λ-calculus without having to explicitly define the compilation from λ-terms to $\mu\tilde{\mu}$-machines 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 truth and falsity value witnesses ($\mathcal{J}(|P|_V)$ and $\mathcal{J}(\|N\|_V)$) which we will define carefully, irrespectively of our compilation scheme – at first. In fact, we will show that we

can *deduce* the compilation function from the dependent typing requirements, starting from the programs of the simplified version – so they need not be fixed in advance.

It is tempting to see truth and falsity justifications as the *erasure* of truth and falsity witnesses: we show in Section 5 that, with some ingenuity, Coq's extraction can indeed be coerced into producing the former from the latter.

4 NORMALIZATION BY REALIZABILITY FOR THE SIMPLY-TYPED λ-CALCULUS

We give a first example of a realizability-based proof of weak normalization of the simply-typed λ-calculus with arrows and sums. Those inductive definitions of syntactic terms, types and type derivations that we gave in Figure 1 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 $\Gamma \vdash t : A$. To help understanding, we stick to the color conventions introduced in Section 2, which take all their meaning here as we combine programming constructs in the object and the meta-langagues: for example, we distinguish the λ-abstraction of the object language $\lambda x.\, t$ and of the meta-language $\lambda \bar{x}.\, \bar{t}$. Again, we will never mix identifiers differing only by their color – you lose little if you do not see the colors.

Recall from Section 2 the definitions of falsity value witnesses for the arrow type

$$\|A \to B\|_V =^{(8)} \{u \cdot e \mid u \in |A|, e \in \|B\|\}$$

as set of co-terms. 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:

$$\mathcal{J}(\|A \to B\|_V) \triangleq \mathcal{J}(|A|) \times \mathcal{J}(\|B\|) \tag{8'}$$

For the sum type, we are naturally lead to define truth value witnesses as

$$|A_1 + A_2|_V \triangleq \{\sigma_i\ t \mid t \in |A_i|\} \qquad (i \in \{1, 2\}) \tag{24}$$

that erases to

$$\mathcal{J}(|A_1 + A_2|_V) \triangleq \mathcal{J}(|A_1|) + \mathcal{J}(|A_2|) \tag{24'}$$

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,

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

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 (which is indeed well-founded, as per our Coq formalization). With this in place, we can now write our simplified adequacy lemma, in the non-dependent version:

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

We will present the code case by case, and discuss why each is well-typed. 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.

$$\mathtt{rea}\quad x\quad \left\{\overline{\Gamma, x:A \vdash x : A}\right\}\quad \bar\rho^{|\Gamma,x:A|}\quad \triangleq\quad \bar\rho(x)^{|A|}$$

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

Abstraction.

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

We have structural information on the return type $\mathcal{J}(|A \to B|) =^{(16')} \mathcal{J}(\|A \to B\|_V{}^{\perp}) =^{(5')} \mathcal{J}(\|A \to B\|_V) \to \mathcal{J}(\bot\!\!\bot)$. It is thus natural to start with a λ-abstraction over $\mathcal{J}(\|A \to B\|_V) =^{(8')} \mathcal{J}(|A|) \times \mathcal{J}(\|B\|)$, matching a pair of an $\bar u : \mathcal{J}(|A|)$ and an $\bar e : \mathcal{J}(\|B\|)$ to return a $\mathcal{J}(\bot\!\!\bot)$.

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 $\mathcal{J}(\bot\!\!\bot)$ by using the auxiliary cut function (22) which boils down to

$$\begin{array}{lcll} \langle\, \_ \mid \_ \,\rangle_A & : & \mathcal{J}(|A|) \to \mathcal{J}(\|A\|) \to \mathcal{J}(\bot\!\!\bot) & \\ \langle\, \bar t \mid \bar e \,\rangle_P & \triangleq & \bar t\ \bar e & (22') \\ \langle\, \bar t \mid \bar e \,\rangle_N & \triangleq & \bar e\ \bar t & \end{array}$$

in the simply-typed setting.

Injections.

$$\mathtt{rea}\quad (\sigma_i\ t)\quad \left\{\frac{\Gamma \vdash t : A_i}{\Gamma \vdash \sigma_i\ t : A_1 + A_2}\right\}\quad \bar\rho\quad \triangleq\quad (\sigma_i\ (\mathtt{rea}\ t\ \bar\rho))^{\perp\perp}\quad (i \in \{1,2\})$$

The recursive call wrapped in the injection function has type $\mathcal{J}(|A_1|) + \mathcal{J}(|A_2|) =^{(24')} \mathcal{J}(|A_1 + A_2|_V)$. This is not the expected type of the `rea` function, which is $\mathcal{J}(|A_1 + A_2|) =^{(15')} \mathcal{J}(|A_1 + A_2|_V{}^{\perp\perp})$. The solution is to appeal to the bi-orthogonal inclusion (23), which boils down to the following (pair of) simply-typed program:

$$\begin{array}{lcllcll} \_^{\perp\perp} & : & \mathcal{J}(|A|_V) \to \mathcal{J}(|A|) & \quad & \_^{\perp\perp} & : & \mathcal{J}(\|A\|_V) \to \mathcal{J}(\|A\|) \\ (\bar v^{|P|_V})^{\perp\perp} & \triangleq & \lambda \bar e^{\|P|}.\, \bar e\ \bar v & & (\bar e^{\|P|_V})^{\perp\perp} & \triangleq & \bar e \\ (\bar t^{|N|_V})^{\perp\perp} & \triangleq & \bar t & & (\bar\pi^{\|N\|_V})^{\perp\perp} & \triangleq & \lambda \bar t^{|N|}.\, \bar t\ \bar\pi \end{array} \qquad (23')$$

Application.

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

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 fact that $|B| =^{(19)} \|B\|_V{}^{\perp}$: the expected type is therefore $\mathcal{J}(\|B\|_V{}^{\perp}) =^{(5')} \mathcal{J}(\|B\|_V) \to \mathcal{J}(\bot\!\!\bot)$.

The function parameter $\bar\pi$ at type $\mathcal{J}(\|B\|_V)$ is injected into $\mathcal{J}(\|B\|)$ by the $\_^{\perp\perp}$ function defined in (23'), and then paired with a recursive call on u at type $\mathcal{J}(|A|)$ to build a $\mathcal{J}(|A|) \times \mathcal{J}(\|B\|) =^{(8')} \mathcal{J}(\|A \to B\|_V)$.

This co-value justification can then be directly applied to the recursive call on t, of type $\mathcal{J}(|A \to B|) =^{(16')} \mathcal{J}(\|A \to B\|_V{}^{\perp}) =^{(5')} \mathcal{J}(\|A \to B\|_V) \to \mathcal{J}(\bot\!\!\bot) =^{(8')} \mathcal{J}(|A|) \times \mathcal{J}(\|B\|) \to \mathcal{J}(\bot\!\!\bot)$.

$$
\begin{array}{llcll}
\mathtt{rea} & x^A & \bar\rho & \triangleq & \bar\rho(x) \\
\mathtt{rea} & (\lambda x^A.\, t^B) & \bar\rho & \triangleq & \lambda(\bar u^{|A|}, \bar e^{\|B\|}).\ \langle\, \mathtt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B \\
\mathtt{rea} & (t^{A\to B}\ u^A) & \bar\rho & \triangleq & \lambda\bar\pi^{\|B\|_V}.\, \mathtt{rea}\ t\ \bar\rho\ (\mathtt{rea}\ u\ \bar\rho, \bar\pi^{\perp\perp}) \\
\mathtt{rea} & (\sigma_i\ t^{A_i}) & \bar\rho & \triangleq & (\sigma_i\ (\mathtt{rea}\ t\ \bar\rho))^{\perp\perp} \qquad (i \in \{1,2\}) \\
\\
\mathtt{rea} & \delta(t^{A_1+A_2}, x_1.\, {u_1}^C, x_2.\, {u_2}^C) & \bar\rho & \triangleq & \\
\multicolumn{5}{l}{\lambda\bar\pi^{\|C\|_V}.\left\langle \mathtt{rea}\ t\ \bar\rho \mid \lambda\bar v^{|A_1+A_2|_V}.\, \mathtt{match}\ \bar v\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ \bar t_1{}^{|A_1|} \to \mathtt{rea}\ u_1\ \bar\rho[x_1 \mapsto \bar t_1]\ \bar\pi \\ \sigma_2\ \bar t_2{}^{|A_2|} \to \mathtt{rea}\ u_2\ \bar\rho[x_2 \mapsto \bar t_2]\ \bar\pi \end{array} \right\rangle_{A_1+A_2}}
\end{array}
$$

Fig. 4. Summary of the simply-typed setting

Case analysis.

$$\mathtt{rea}\ \delta(t, x_1.\, u_1, x_2.\, u_2)\ \left\{ \frac{\Gamma \vdash t : A_1 + A_2 \qquad \Gamma, x_i : A_i \vdash u_i : C}{\Gamma \vdash \delta(t, x_1.\, u_1, x_2.\, u_2) : C} \right\}\ \bar\rho\ \triangleq$$

$$\lambda\bar\pi^{\|C\|_V}.\left\langle \mathtt{rea}\ t\ \bar\rho \mid \lambda\bar v^{|A_1+A_2|_V}.\, \mathtt{match}\ \bar v\ \mathtt{with} \;\middle|\; \begin{array}{l} \sigma_1\ \bar t_1{}^{|A_1|} \to \mathtt{rea}\ u_1\ \bar\rho[x_1 \mapsto \bar t_1]\ \bar\pi \\ \sigma_2\ \bar t_2{}^{|A_2|} \to \mathtt{rea}\ u_2\ \bar\rho[x_2 \mapsto \bar t_2]\ \bar\pi \end{array} \right\rangle_{A_1+A_2}$$

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 $\mathcal{J}(\bot\!\!\bot)$: we can cut 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 $\mathcal{J}(\bot\!\!\bot)$, using the cut function: $\left\langle\, \bar t \mid \lambda x.\, M^{\bot\!\!\bot} \,\right\rangle_A : \mathcal{J}(\bot\!\!\bot)$. (If you know about monads, this is the *bind* operation.)

This concludes our implementation of the adequacy lemma. For the sake of completeness, its complete definition is summarized in Figure 4. 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 $\mathtt{rea}\ t\ \bar\rho$ returns a function type, so it may not evaluate its argument immediately. We further dwell on this question in Section 6, in which we present a systematic study of the various design choices available in the realizability proof. As for now, we move from the simplified, non-dependent types to the more informative dependent types. This calls for defining the compilation from λ-terms to $\mu\tilde\mu$-machines, which has been left unspecified so far.

5 A DEPENDENTLY-TYPED REALIZABILITY PROGRAM

In the following, we undo the simplification presented in Section 3.3, by moving back to dependent types. The best way to make sure that our program is type-correct is to run a type-checker on it. To this end, this section has been developed under the scrutiny of the Coq proof assistant. Working with Coq offers two advantages. First, it has allowed us to interactively explore the design space in a type-driven manner, using type-level computation to assist this exploration. Second and unlike the set-theoretic presentation, we can precisely delineate the computational content of our proof from its propositional one.

We begin by refining our set-theoretic statement of the realizability lemma

$$\mathtt{rea} : \forall\{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ \{\Gamma \vdash t : A\}\ \to \rho \in |\Gamma| \to \lfloor t \rfloor[\rho] \in |A|$$

into dependent type theory. We want to be able to extract a realizer (a $\mu\tilde{\mu}$ term) from a typing derivation. This is achieved by a function

$$\lfloor - \rfloor : \forall \{\Gamma\}\ t\ \{A\}.\ \{\Gamma \vdash t : A\} \to \mathbb{T}$$

Then, we must set up the interpretation of types. We notice that the pole we are interested in, defined by

$$m \in \perp\!\!\!\perp \triangleq \{m_n \in \mathbb{M}_N \mid m \rightsquigarrow m_1 \rightsquigarrow \ldots \rightsquigarrow m_n\}$$

is rather distinctive: it consists in a computationally-relevant object (the normal form $m_n \in \mathbb{M}_N$) as well as a proposition stating that the normal form is indeed related to the original program. We would like to exploit Coq's extraction mechanism [Letouzey 2008] to obtain the simply-typed program of Section 4 by extracting the adequacy lemma to OCaml.

We thus define the class of predicates over A

$$\mathsf{Fam}_{Rel}\ (A : \mathsf{Type}) \triangleq \begin{cases} \mathcal{J} : \mathsf{Type} \\ \mathcal{R} : A \to \mathcal{J} \to \mathsf{Prop} \end{cases}$$

for which the first component is computationally relevant while the second component is erasable (*i.e.*, of sort Prop in Coq). Given a predicate $P : \mathsf{Fam}_{Rel}\ A$, we write $\mathcal{J}(P)$ its first component while we write $\mathcal{R}(P)$ its second component. An element $a : A$ is said to verify P if we have

$$a \in P \triangleq \{a_0 : \mathcal{J}(P) \mid \mathcal{R}(P)\ a\ a_0\} \tag{25}$$

Note that we use a subset type (written $\{\_ \mid \_\}$) here: set membership erases to the underlying witness, here a_0, throwing away its logical justification. The pole is an example of such a predicate:

$$\begin{aligned} &\perp\!\!\!\perp\ :\ \mathsf{Fam}_{Rel}\ \mathbb{M} \\ &\perp\!\!\!\perp \triangleq \begin{cases} \mathcal{J} \triangleq \mathbb{M}_N \\ \mathcal{R} \triangleq \lambda m.\, \lambda m_n.\, m \rightsquigarrow \ldots \rightsquigarrow m_n \end{cases} \end{aligned} \tag{21"}$$

Besides, taking the orthogonal of a predicate in $S : \mathsf{Fam}_{Rel}\ \mathbb{T}$ yields a predicate in $\mathsf{Fam}_{Rel}\ \mathbb{E}$:

$$\begin{aligned} & (\lambda t.\, t \in S)^{\perp} \\ \cong^{(4)}\ & \lambda e.\, \forall \{t : \mathbb{T}\}.\, t \in S \to \langle\, t \mid e \,\rangle \in \perp\!\!\!\perp \\ =^{(21")}\ & \lambda e.\, \forall \{t : \mathbb{T}\}.\, t \in S \to \{m_n : \mathcal{J}(\perp\!\!\!\perp) \mid \mathcal{R}(\perp\!\!\!\perp)\ \langle\, t \mid e \,\rangle\ m_n\} \\ =^{\text{choice}}\ & \lambda e.\, \{\bar{e} : \forall \{t : \mathbb{T}\}.\, t \in S \to \mathcal{J}(\perp\!\!\!\perp) \mid \forall \{t : \mathbb{T}\}.\, (\bar{t} : t \in S) \to \mathcal{R}(\perp\!\!\!\perp)\ \langle\, t \mid e \,\rangle\ (\bar{e}\ \bar{t})\} \\ =^{(25)}\ & \lambda e. \begin{cases} \mathcal{J} \triangleq \forall \{t : \mathbb{T}\}.\, t \in S \to \mathcal{J}(\perp\!\!\!\perp) \\ \mathcal{R} \triangleq \lambda e.\, \lambda \bar{e}.\, \forall \{t : \mathbb{T}\}.\, (\bar{t} : t \in S) \to \mathcal{R}(\perp\!\!\!\perp)\ \langle\, t \mid e \,\rangle\ (\bar{e}\ \bar{t}) \end{cases} \end{aligned}$$

So, we read off the definition of the orthogonal from this isomorphism, and symmetrically when going from co-terms to terms along the specification (5):

$$\begin{aligned} |A|^{\perp}\ &:\ \mathsf{Fam}_{Rel}\ \mathbb{E} \\ |A|^{\perp}\ &\triangleq \begin{cases} \mathcal{J} \triangleq \forall \{t : \mathbb{T}\}.\, t \in |A| \to \mathcal{J}(\perp\!\!\!\perp) \\ \mathcal{R} \triangleq \lambda e.\, \lambda \bar{e}.\, \forall \{t : \mathbb{T}\}.\, (\bar{t} : t \in |A|) \to \mathcal{R}(\perp\!\!\!\perp)\ \langle\, t \mid e \,\rangle\ (\bar{e}\ \bar{t}) \end{cases} \\ \|A\|^{\perp}\ &:\ \mathsf{Fam}_{Rel}\ \mathbb{T} \\ \|A\|^{\perp}\ &\triangleq \begin{cases} \mathcal{J} \triangleq \forall \{e : \mathbb{E}\}.\, e \in \|A\| \to \mathcal{J}(\perp\!\!\!\perp) \\ \mathcal{R} \triangleq \lambda t.\, \lambda \bar{t}.\, \forall \{e : \mathbb{E}\}.\, (\bar{e} : e \in \|A\|) \to \mathcal{R}(\perp\!\!\!\perp)\ \langle\, t \mid e \,\rangle\ (\bar{t}\ \bar{e}) \end{cases} \end{aligned}$$

Assuming that (co-)term witnesses are predicates of that form, the value witnesses for sum and arrow can naturally be expressed as elements of $\mathsf{Fam}_{Rel}(-)$: exactly as in Section 4, the underlying type consists, in both cases, in a pair of meta-evaluators while the attached relation asserts the

validity of each evaluator. Thus, the set-theoretic definitions (24) and (8) of value witnesses for arrows and sums translates into

$$\|A \to B\|_V : \mathtt{Fam}_{Rel}\ \mathbb{E}$$

$$\|A \to B\|_V \triangleq \left\{ \begin{array}{l} \mathcal{J} \triangleq \mathcal{J}(|A|) \times \mathcal{J}(\|B\|) \\ \mathcal{R} \triangleq \lambda t.\, \lambda \bar{t}.\, \mathtt{match}\ t, \bar{t}\ \mathtt{with} \end{array} \left| \begin{array}{rl} (u \cdot e), (\bar{u}, \bar{e}) & \to \mathcal{R}(|A|)\ u\ \bar{u}\ \wedge \mathcal{R}(\|B\|)\ e\ \bar{e} \\ \_ & \to \bot \end{array} \right. \right. \tag{8"}$$

$$|A_1 + A_2|_V : \mathtt{Fam}_{Rel}\ \mathbb{T}$$

$$|A_1 + A_2|_V \triangleq \left\{ \begin{array}{l} \mathcal{J} \triangleq \mathcal{J}(|A_1|) + \mathcal{J}(|A_2|) \\ \mathcal{R} \triangleq \lambda t.\, \lambda \bar{t}.\, \mathtt{match}\ t, \bar{t}\ \mathtt{with} \end{array} \left| \begin{array}{rl} \sigma_1\ u, \sigma_1\ \bar{u} & \to \mathcal{R}(|A_1|)\ u\ \bar{u} \\ \sigma_2\ u, \sigma_2\ \bar{u} & \to \mathcal{R}(|A_2|)\ u\ \bar{u} \\ \_ & \to \bot \end{array} \right. \right. \tag{24"}$$

We tie the knot by recursion over types and obtain a mutually recursive pair of functions from syntactic types of the object language to predicates in our meta-language: a truth witness interpretation targeting $\mathtt{Fam}_{Rel}\ \mathbb{T}$ and a falsity witness interpretation targeting $\mathtt{Fam}_{Rel}\ \mathbb{E}$. This naturally extends to contexts.

$$\|P\|, \|N\| : \mathtt{Fam}_{Rel}\ \mathbb{E} \qquad\qquad |P|, |N| : \mathtt{Fam}_{Rel}\ \mathbb{T}$$

$$\|P\| \triangleq |P|_V{}^{\perp} \tag{13"} \qquad\qquad |P| \triangleq |P|_V{}^{\perp\perp} \tag{15"}$$

$$\|N\| \triangleq \|N\|_V{}^{\perp\perp} \tag{14"} \qquad\qquad |N| \triangleq \|N\|_V{}^{\perp} \tag{16"}$$

The adequacy lemma thus becomes

$$\mathtt{rea} : \forall \{\Gamma\}\ t\ \{A\}\ \{\rho\}.\ \{\Gamma \vdash t : A\}\ \to \rho \in |\Gamma| \to \lfloor t \rfloor [\rho] \in |A|$$

where $\lfloor t \rfloor$ compiles a term of the object language into a $\mu\tilde{\mu}$ term in the meta-language that witnesses the reduction. Rather than define this function upfront, we are going to reverse-engineer it from the proof of the adequacy lemma. Indeed, the typing constraints of the dependent version force us to exhibit a reduction sequence, that is, a suitable compilation function.

The dependent version of the adequacy lemma has the same structure as the non-dependent one. Apart from enriching types, the main difference is that we justify the existence of the desired reduction sequence through computationally transparent annotations tracking the use of reduction rules.

Abstraction. Consider the λ-abstraction case of Section 4 (p.14):

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

We transform this code to a dependently-typed case, by adding annotations, leaving the code structure otherwise unchanged:

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

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

Indeed, by definition (8") of $\|A \to B\|_V$, we can abstract over a pair in the object language $\lambda(u \cdot e : \mathbb{E})$ instead of the more general $\lambda\,\{(e_0 : \mathbb{E})\}$ since the pattern matching implicit in the subsequent abstraction necessarily teaches us that e_0 must be equal to a pair, so any other case would be absurd: matching on a pair covers all non-trivial 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. This gives to the expression $\langle\, \mathtt{rea}\ t\ \bar\rho[x \mapsto \bar u] \mid \bar e \,\rangle_B$ the type $\langle\, \lfloor t \rfloor[\rho, x \mapsto u] \mid e \,\rangle \in \perp\!\!\!\perp$. But we just abstracted on $u \cdot e \in \|A \to B\|_V$, in order to form a complete term of type $\lfloor \lambda x.\, t \rfloor[\rho] \in \|A \to B\|_V{}^{\perp}$ so the expected type is in fact $\langle\, \lfloor \lambda x.\, t \rfloor[\rho] \mid u \cdot e \,\rangle \in \perp\!\!\!\perp$. To address this issue, we define an auxiliary function $\_^{\mathtt{lam}}$ of type

$$\_^{\mathtt{lam}} : \forall\{t\ \rho\ x\ u\ e\}.\ \langle\, t[\rho, x \mapsto u] \mid e \,\rangle \in \perp\!\!\!\perp \to \langle\, \lfloor \lambda x.\, t \rfloor[\rho] \mid u \cdot e \,\rangle \in \perp\!\!\!\perp$$

This auxiliary function exactly corresponds to the fact that the pole is closed under anti-reduction of machine configurations (Figure 2). Indeed, the required reduction

$$\langle\, \lfloor \lambda x.\, t \rfloor \mid u \cdot e \,\rangle \rightsquigarrow \langle\, t[x \mapsto u] \mid e \,\rangle$$

is a consequence of the general rule for functions

$$\langle\, \mu(x \cdot \alpha).\, \langle\, t \mid \alpha \,\rangle \mid u \cdot e \,\rangle \rightsquigarrow \langle\, t[x \mapsto u, \alpha \mapsto e] \mid e \,\rangle$$

plus the fact that translations of λ-terms $\lfloor t \rfloor$ have no free co-term variable α – as can be proved by direct induction. Note that if we had *not* given you the definition of $\lfloor \lambda x.\, t \rfloor$, you would be *asked* to pick $\mu(x \cdot \alpha).\, \langle\, t \mid \alpha \,\rangle$ from the type-checking constraints that arise in the proof.

To define this function, recall our characterization of the pole:

$$m \in \perp\!\!\!\perp \quad \cong \quad \{m_n \in \mathbb{M}_N \mid m \rightsquigarrow m_1 \rightsquigarrow \ldots \rightsquigarrow m_n\}$$

The function $\_^{\mathtt{lam}}$ takes a normal form for the reduced machine, and has to return a normal form for the not-yet-reduced machines; this is the exact same normal form, with an extra reduction step.

$$\begin{aligned}\_^{\mathtt{lam}}\ \{t\ \rho\ x\ u\ e\} \quad \triangleq \quad & \lambda\{m_n \mid \langle\, \lfloor t \rfloor[x \mapsto u] \mid e \,\rangle \rightsquigarrow \ldots \rightsquigarrow m_n\}. \\ & \{m_n \mid \langle\, \lfloor \lambda x.\, t \rfloor \mid u \cdot e \,\rangle \rightsquigarrow \langle\, \lfloor t \rfloor[x \mapsto u] \mid e \,\rangle \rightsquigarrow \ldots \rightsquigarrow m_n\}\end{aligned}$$

Application. Similarly, the application case can be dependently typed as

$$\mathtt{rea}\ (t^{A \to B}\ u^t)\ \bar\rho \ \triangleq\ \lambda\{\pi : \mathbb{E}\}.\, \lambda\bar\pi : \pi \in \|B\|_V.\, \left(\mathtt{rea}\ t\ \bar\rho\ \left(\mathtt{rea}\ u\ \bar\rho, \bar\pi^{\perp\perp}\right)\right)^{\mathtt{app}}$$

The function $\_^{\perp\perp}$, which we had defined 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 $\mathtt{rea}\ t\ \bar\rho\ \left(\mathtt{rea}\ u\ \bar\rho, \bar\pi^{\perp\perp}\right)$ has type $\langle\, \lfloor t \rfloor[\rho] \mid \lfloor u \rfloor[\rho] \cdot \pi \,\rangle \in \perp\!\!\!\perp$. For the whole term to be well-typed, the auxiliary function $\_^{\mathtt{app}}$ needs to have the type

$$\forall\{t\ u\ \rho\ \pi\}.\, \langle\, \lfloor t \rfloor[\rho] \mid \lfloor u \rfloor[\rho] \cdot \pi \,\rangle \in \perp\!\!\!\perp \to \langle\, \lfloor t\ u \rfloor[\rho] \mid \pi \,\rangle \in \perp\!\!\!\perp$$

Again, this is exactly an anti-reduction rule; not as a coincidence, but as a direct result of the translation of application in $\mu\tilde\mu$: $\lfloor t\ u \rfloor \triangleq \mu\alpha.\, \langle\, \lfloor t \rfloor \mid \lfloor u \rfloor \cdot \alpha \,\rangle$. The definition is similar to $\_^{\mathtt{lam}}$.

Case analysis. The dependently-typed treatment of case analysis involves three decorations, $\_^{\mathtt{inj}_1}$, $\_^{\mathtt{inj}_2}$ and $\_^{\mathtt{case}}$:

$$\begin{aligned}&\mathtt{rea}\ \delta(t^{A_1+A_2}, x_1.\, u_1{}^C, x_2.\, u_2{}^C)\ \bar\rho \ \triangleq \\ &\quad \lambda\{\pi : \mathbb{E}\}.\, \lambda\bar\pi : \pi \in \|C\|_V. \\ &\quad \left\langle\, \mathtt{rea}\ t\ \bar\rho \mid \lambda\{v\}.\, \lambda\bar v : v \in |A_1 + A_2|_V.\ \left|\begin{array}{l} \mathtt{match}\ v, \bar v\ \mathtt{with} \\ \sigma_1\ t_1, \sigma_1\ \bar t_1 \to (\mathtt{rea}\ u_1\ \bar\rho[x_1 \mapsto \bar t_1]\ \bar\pi)^{\mathtt{inj}_1} \\ \sigma_2\ t_2, \sigma_2\ \bar t_2 \to (\mathtt{rea}\ u_2\ \bar\rho[x_2 \mapsto \bar t_2]\ \bar\pi)^{\mathtt{inj}_2} \end{array}\right. \right\rangle_{A_1+A_2}^{\mathtt{case}}\end{aligned}$$

Let us show how the typing constraints in this term let us deduce/recover exactly the translation we gave in Figure 3 of sum elimination into $\mu\tilde\mu$.

From the outer typing constraint, we know that the result type of $\_^{\mathtt{case}}$ must be of type $\langle\, \lfloor \delta(t, x_1.\, u_1, x_2.\, u_2) \rfloor[\rho] \mid \pi \,\rangle \in \perp\!\!\!\perp$. Its argument is of the form $\langle\, \mathtt{rea}\ t\ \rho \mid \ldots \,\rangle_{A_1+A_2}$, so $\_^{\mathtt{case}}$ must perform an anti-reduction argument on a reduction step of the form

$$\langle\, \lfloor \delta(t, x_1.\, u_1, x_2.\, u_2) \rfloor \mid \pi \,\rangle \rightsquigarrow \langle\, \lfloor t \rfloor \mid \ldots \,\rangle$$

Let us write $(\lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi)$ for the co-term in the right-hand-side of the reduction rule above. From the desired reduction step above, we know that we need the translation

$$\lfloor \delta(t, x_1.\, u_1, x_2.\, u_2) \rfloor \triangleq \mu\alpha.\, \langle\, t \mid (\lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \alpha) \,\rangle$$

where $(\lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \alpha)$ remains to be determined.

As a second step, we look at the typing constraints for the inner auxiliary functions $\_^{\mathtt{inj}_i}$, in the terms $(\mathtt{rea}\ u_i\ \bar{\rho}[x_i \mapsto \bar{t}_i]\ \bar{\pi})^{\mathtt{inj}_i}$. These terms must prove that the co-term $(\lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi)$ is in $\|A_1 + A_2\|$, that is, that putting it against any *constructor* $\sigma_i\ t_i$ goes in the pole. The result type of $\_^{\mathtt{inj}_i}$ is thus of the form

$$\langle\, \sigma_i\ t_i \mid \lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi \,\rangle \in \perp\!\!\!\perp$$

Finally, we know the input type of $\_^{\mathtt{inj}_i}$, which is of the form $\langle\, u_i[x_i/t_i] \mid \pi \,\rangle \in \perp\!\!\!\perp$ so we know that this function must correspond to an anti-reduction argument for the reduction

$$\langle\, \sigma_i\ t_i \mid \lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi \,\rangle \rightsquigarrow \langle\, u_i[x_i/t_i] \mid \pi \,\rangle \in \perp\!\!\!\perp$$

which forces us to pose:[8]

$$(\lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi) \triangleq \tilde{\mu}\, [x_1.\, \langle\, u_1 \mid \pi \,\rangle \mid x_2.\, \langle\, u_2 \mid \pi \,\rangle]$$

which gives, indeed, the same definition as Figure 3:

$$\begin{aligned} \lfloor \delta(t, x_1.\, u_1, x_2.\, u_2) \rfloor &= \langle\, t \mid \lfloor \delta(\Box, x_1.\, u_1, x_2.\, u_2) \rfloor ; \pi \,\rangle \\ &= \langle\, t \mid \tilde{\mu}\, [x_1.\, \langle\, u_1 \mid \pi \,\rangle \mid x_2.\, \langle\, u_2 \mid \pi \,\rangle] \,\rangle \end{aligned}$$

Sum injections. Injections, being computationally inert, are straightforward to handle: an injection in the object language compiles into a $\mu\tilde{\mu}$ injection and the corresponding realizability program does not require decoration since it does not need to track a reduction of the configuration.

Wrapping up. We would like to point out that all of the compilation rules from the λ-calculus to our abstract machines could be *recovered* by studying the typing constraints in our proof.

It means that this compilation scheme is, 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 – which, in our setting, correspond to those co-terms that belong not only to $\|N\|$, but also to $\|N\|_V$.

Mechanized formalization in Coq. Our formalization covers more type constructors than we had space to present here, handling both (positive) natural numbers and positive products[9]. While positive products are handled along the same line as sums, interpreting the elimination form for natural numbers lead us naturally to introduce a standalone lemma that extracts (somewhat to our disbelief) to an iterator over integers. This phenomena by which we "coincidentally extract to the right program" is noteworthy. In fact, there is little to no choice in writing the proof, once we get familiarized with the mechanics of our dependent types.

As a result of our careful segregation of the computational and propositional content of our definitions, the dependently-typed adequacy lemma – proved using tactics and concluded with the vernacular `Defined.` – admits a straightforward extraction: the second component of the subset types are erased, leaving only the normal forms. The resulting OCaml program is almost literally the program written in Section 4, provided that we ignore the remains of dependent types that

[8] We took the idea of seeing introduction of a new pattern-matching form as the resolutions of some equations from Munch-Maccagnoni [2013].

[9] In intuitionistic logic, there is little difference between the positive and the negative product, so a product is always suspect of being negative type in hiding. The elimination form of positive products matches strictly on both arguments whereas negative products are eliminated by two projections. The difference is glaring in the presence of side-effects.

extraction fails to eliminate, such as terms, stacks, and substitutions over them. In particular, we uncovered during this process that the reduction sequence is justified by appeal to computationally irrelevant annotations translating the anti-reduction closure of the pole, a reassuring and satisfying result. An informal dead-code elimination argument suggests that we could safely remove the remains of dependent types but convincing Coq that is safe to do so remains future work.

To illustrate the fact that the computational content of the adequacy lemma does not depend on a particular choice of the pole, the proof itself is parametrized by a Coq module specifying the computational and logical content of a pole. We provide two instantiations of pole to support extraction: full machine configurations and natural numbers (suited for configurations that normalize to integers). In this way, we are able to run several examples of derivations in the simply-typed λ-calculus through the adequacy lemma and extract the corresponding normalized configuration or integer. The supplementary material provides information on how to run with these proofs/programs.

6 ALTERNATIVE REALIZATIONS

We made two arbitrary choices when we decided to define $\|A \to B\|_V \triangleq \{u \cdot e \mid u \in |A|, e \in \|B\|\}$. There are in fact four possibilities with the same structure:

(1) $\{u \cdot e \mid u \in |A|, e \in \|B\|\}$ **(2)** $\{u \cdot e \mid u \in |A|, e \in \|B\|_V\}$

(3) $\{u \cdot e \mid u \in |A|_V, e \in \|B\|\}$ **(4)** $\{u \cdot e \mid u \in |A|_V, e \in \|B\|_V\}$

In our Coq formalization, we study the four possibilities and see that they all work but each of them gives a slightly different realization program. For conciseness, we will only treat Variant **(3)** in this section and omit sum types from these variants; it can be handled separately (and would also give rise to four different choices), as we did in our formalization.

We will not repeat the reverse engineering process of the previous section, but directly jump to the conclusion of what translation to (arrow-related) co-terms they suggest, and which reduction strategy they must follow. We observe that the first two variants correspond to a call-by-name evaluation strategy, while the two latter correspond to call-by-value; forcing us, in particular, to use the $\tilde{\mu}x.m$ construction [Curien and Herbelin 2000] in the compilation to $\mu\tilde{\mu}$ terms.

In order to distinguish the compilation functions of the various variants, we mark them with their variant number as exponent: $\lfloor t \rfloor^i$ for variant **(i)**.

Variant (1) $\{u \cdot e \mid u \in |A|, e \in \|B\|\}$. This interpretation corresponds to the realization program presented in Section 4, recalled here for ease of comparison:

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

$$
\begin{array}{rl}
\lfloor x \rfloor^1 & \triangleq x \\
\lfloor \lambda x.\, t \rfloor^1 & \triangleq \mu(x \cdot \alpha).\, \langle\, \lfloor t \rfloor^1 \mid \alpha \,\rangle \\
\lfloor t\ u \rfloor^1 & \triangleq \mu\alpha.\, \langle\, \lfloor t \rfloor^1 \mid \lfloor u \rfloor^1 \cdot \alpha \,\rangle
\end{array}
$$

Variant (3) $\{u \cdot e \mid u \in |A|_V, e \in \|B\|\}$. A notable difference in this variant – and Variant **(4)** – 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 $t : A$ 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.

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

Had we kept $\bar{\rho} : \rho \in |\Gamma|$, we would have only $\bar{\rho}(x)$ in the variable case, but $\bar{\rho}[x \mapsto \bar{v}^{\perp\perp}]$ 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)**:

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

The relation between Variant **(1)** and Variant **(3)** seems to be an inlining of $\mathtt{rea}\ u\ \bar{\rho}$, yet the shapes of the terms hint at a distinction between call-by-name and call-by-value evaluation. These two versions are equivalent whenever A is negative, as in that case the cut $\langle\, \mathtt{rea}\ u\ \bar{\rho} \mid \lambda\bar{v}_u. \ldots \rangle_A$ applies its left-hand side as an argument to its right-hand side, giving exactly the definition of Variant **(1)** after β-reduction. However, once again, considering the dependent version forces us to clarify the evaluation dynamics:

$$
\begin{array}{l}
\mathtt{rea}\ \{\Gamma\}\ (t^{A\to B}\ u^A)\ \{\rho\}\ (\bar{\rho} : \rho \in |\Gamma|_V) \triangleq \lambda\{\pi : \mathbb{E}\}.\, \lambda\bar{\pi} : \pi \in \|B\|_V. \\
\quad \Big\langle\, \mathtt{rea}\ u\ \bar{\rho} \mid \lambda\{v_u : \mathbb{T}\}.\, \lambda(\bar{v}_u : v_u \in |A|_V).\ \big\langle\, \mathtt{rea}\ t\ \bar{\rho} \mid (\bar{v}_u, \bar{\pi}^{\perp\perp})\, \big\rangle_{A\to B}^{\mathsf{app-body}}\, \Big\rangle_A^{\mathsf{app-arg}}
\end{array}
$$

The annotation $\_^{\mathsf{app-body}}$ corresponds to a reduction to the configuration $\langle\, \lfloor t \rfloor^3[\rho] \mid v_u \cdot \pi\, \rangle$. Its return type – which determines the configuration 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 $\lfloor u \rfloor^3[\rho] \in |A|$ on the left, and thus on the right an $e \in \|A\|$ for some co-term e. By definition of $e \in \|A\|$, the return type of $\_^{\mathsf{app-body}}$ should thus be of the form $\langle\, v_u \mid e\, \rangle \in \perp\!\!\!\perp$.

What could be the result of $\lfloor u \rfloor^3$ that would respect the reduction "equation" $\langle\, v_u \mid \lfloor u \rfloor^3\, \rangle \rightsquigarrow \langle\, \lfloor u \rfloor^3[\rho] \mid v_u \cdot \pi\, \rangle$ so that $\_^{\mathsf{app-body}}$ amounts to stability under anti-reduction? Using $\mu\tilde{\mu}$'s $\tilde{\mu}$ binder [Curien and Herbelin 2000], we define $\lfloor u \rfloor^3 \triangleq \tilde{\mu}v_x.\, \langle\, t[\rho] \mid v_x \cdot \pi\, \rangle$ which is subject to the desired reduction: $\langle\, v \mid \tilde{\mu}x.\, m\, \rangle \rightsquigarrow m[x \mapsto v]$.

From here, the input and output type of $\_^{\mathsf{app-arg}}$ are fully determined, allowing to state a final equation: $\langle\, \lfloor t\ u \rfloor^3 \mid \pi\, \rangle \rightsquigarrow \langle\, \lfloor u \rfloor^3 \mid \tilde{\mu}v_u.\, \langle\, \lfloor t \rfloor^3 \mid v_u \cdot \pi\, \rangle\, \rangle$ that is solved by taking $\lfloor t\ u \rfloor^3 \triangleq \mu\alpha.\, \langle\, \lfloor u \rfloor^3 \mid \tilde{\mu}v_u.\, \langle\, \lfloor t \rfloor^3 \mid v_u \cdot \alpha\, \rangle\, \rangle$. Summing up:

$$
\begin{array}{rl}
\lfloor x \rfloor^3 & \triangleq x \\
\lfloor \lambda x.\, t \rfloor^3 & \triangleq \mu(x \cdot \alpha).\, \langle\, \lfloor t \rfloor^3 \mid \alpha\, \rangle \\
\lfloor t\ u \rfloor^3 & \triangleq \mu\alpha.\, \langle\, \lfloor u \rfloor^3 \mid \tilde{\mu}v_u.\, \langle\, \lfloor t \rfloor^3 \mid v_u \cdot \alpha\, \rangle\, \rangle
\end{array}
$$

which corresponds to the compilation scheme from the call-by-value λ-calculus into $\mu\tilde{\mu}$.

A closer look at reductions. Looking at these two variants, the code itself may not be clear enough to infer their evaluation order. However, their compilation to $\mu\tilde{\mu}$ machines, 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 (Variant **(3)**), while just requiring arbitrary witnesses gives us call-by-name transitions (Variant **(1)**).

Finally, an interesting point 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 sum type. Any of these choices for $\|A \to B\|_V$ may be combined with any choice for $|A + B|_V$ (basically, lazy or strict sums; the Coq formalization has a variant with lazy sums) to form a complete proof[10]. We see this as yet another manifestation of the claimed "modularity" of realizability and logical-relation approaches, which allows studying connectives independently from each other – once a notion of computation powerful enough to support them all has been fixed.

7 RELATED WORK

Classical realizability. Realizability has been used by logicians to enrich a logic with new axioms and constructively justify such extensions. To do so, one extends 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. For example, extending the base calculus with a suitably-defined `callcc` construct provides a truth witness for Pierce's law (and thus, provides a computational justification to the addition of the principle of excluded middle in the source logic). When possible, this gives us a deeper (and often surprising) understanding of logical axioms. This paper takes the opposite approach, starting from a logic (embodied by the simply-typed λ-calculus) and an interpretation of its typed values to re-discover the computational content of its realizers.

Intuitionistic realizability. Oliva and Streicher [2008] factor the use of classical realizability for a negative fragment of second-order logic as the composition of intuitionistic realizability after a CPS translation. We know that translating λ-terms into abstract machines then back corresponds to a CPS transform. In this work, the authors remark that classical realizability corresponds to translating the abstract machines back into (CPS-forms of) λ-terms, then doing intuitionistic realizability on the resulting terms. The bi-orthogonal closure arises from the intuitionistic realizability interpretation of the double-negation translation of intuitionistic formulas (the types of terms in the original machine).

From the point of view of a user wishing to understand how realizability techniques prove normalization of a given well-typed term, there are thus two choices: either see this term as part of an abstract machine language, and look at the classical realizability proof on top of it, or look at the CPS translation of this term and look at the intuitionistic realizability proof on top of it. Oliva and Streicher show that those two interpretations agree; the question is then to know which will give the better explanation. In our opinion, the abstract machine approach, by giving a direct syntax for continuations instead of a functional encoding, make it easier to follow what is going on. In particular, the justification of the difference in treatment between negative and positive types seems particularly clean in such a symmetric setting; it is also easy to see the relation between changes to the configuration reductions and the reduction strategy of the proof. This is precisely the term-level counterpart of the idea that sequent calculus is more convenient than natural deduction to reason about cut-elimination in presence of positives. That said, because our meta-language is also a typed λ-calculus, some aspects of the CPS-producing transformation from machines to λ-terms are also present in our system.

Ilik [2013] 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

[10] 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.

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 (~ 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.

Normalization by Evaluation. There are evidently strong links with normalization by evaluation (NbE). The previously cited work of Ilik [2013] 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 exhibit any relation between the input and output term.

System L. It is not so surprising that the evaluation function exhibited in Section 4 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, Munch-Maccagnoni [2009] shows that it is a good calculus to study classical realizability. 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's PhD thesis [Munch-Maccagnoni 2013].

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 [Herbelin 2011]. 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).

Another, more compositional way to understand the results presented in this paper consists in performing a *typed* translation from the λ-calculus to typed System L, where we have a typing judgment for terms of the form $\Gamma \vdash t : A$, a typing judgment for co-terms of the form $\Gamma \mid e : A \vdash A$ and well-typed configurations $m : \Gamma \vdash$ are the pair of a well-typed term, and a well-typed co-term, interacting along the same type A. This translation amounts to a CPS transform, System L being essentially a direct syntax for writing programs in CPS.

This type system is also backed by an adequacy lemma 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 m and ρ such as $m : \Gamma \vdash$ and $\rho \in |\Gamma|$, we have $m[\rho] \in \perp\!\!\!\perp$

A careful translation from the λ-calculus to System L would have allowed us to piggy-back on the adequacy lemma of System L to obtain the desired adequacy lemma on λ-terms. In fact, we

saw in Section 6 that our formalization is currently organized around a choice of translation from the λ-terms to $\mu\tilde{\mu}$: the missing piece consists in going from $\mu\tilde{\mu}$ to System L, which amounts to specifying the polarity of our translation – or, put otherwise, deciding on the strictness/laziness of the constructs of the language. Such a translation goes beyond the intent of the present paper, which is meant as providing a slow-paced transition from approaches based on λ-calculus to ones based on abstract machines in meta-theoretical works.

Realizability in PTS. Bernardy and Lasson [2011] 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., $[\![t\ u]\!] \triangleq \mathsf{app}([\![t]\!], [\![u]\!])$) given enough auxiliary functions.

Conclusion

At which point in a classical realizability proof is the "real work" done? We have seen that the computational content of 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. 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`.

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.

ACKNOWLEDGMENTS

We were fortunate to discuss this work with Guillaume Munch-Maccagnoni, Jean-Philippe Bernardy, Hugo Herbelin and Marc Lasson, who were patient and pedagogical in their explanations. We furthermore thank Adrien Guatto, Guillaume Munch-Maccagnoni, Étienne Miquey, Philip Wadler and anonymous reviewers for their feedback on the article.

REFERENCES

Andreas Abel. 2013. *Normalization by Evaluation: Dependent Types and Impredicativity.* Ph.D. Dissertation. Ludwig-Maximilians-Universität München. Habilitation thesis.

Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. 2013. Copatterns: programming infinite structures by observations. In *POPL*.

Jean-Marc Andreoli. 1992. Logic Programming with Focusing Proofs in Linear Logic. *Journal of Logic and Computation* 2, 3 (1992), 297–347. https://doi.org/10.1093/logcom/2.3.297

Robert Atkey, Neil Ghani, and Patricia Johann. 2014. A relationally parametric model of dependent type theory. In *POPL*. https://doi.org/10.1145/2535838.2535852

Henk Barendregt and Giulio Manzonetto. 2013. Turing's contributions to lambda calculus. In *Alan Turing – His Work and Impact*.

Jean-Philippe Bernardy and Marc Lasson. 2011. Realizability and Parametricity in Pure Type Systems. In *FoSSaCS*. https://doi.org/10.1007/978-3-642-19805-2_8

Aloïs Brunel. 2014. *The monitoring power of forcing program transformations.* Ph.D. Dissertation. Université Paris 13. https://hal.archives-ouvertes.fr/tel-01162997

William R. Cook. 2009. On understanding data abstraction, revisited. In *OOPSLA*.

Pierre-Louis Curien, Marcelo P. Fiore, and Guillaume Munch-Maccagnoni. 2016. A theory of effects and resources: adjunction models and polarised calculi. In *POPL*. https://doi.org/10.1145/2837614.2837652

Pierre-Louis Curien and Hugo Herbelin. 2000. The duality of computation. In *ICFP*. https://doi.org/10.1145/357766.351262

Olivier Danvy. 2006. *An Analytical Approach to Program as Data Objects.* Ph.D. Dissertation.

Paul Downen and Zena M. Ariola. 2018. A tutorial on computational classical logic and the sequent calculus. *J. Funct. Program.* (2018).

Paul Downen, Philip Johnson-Freyd, and Zena M. Ariola. 2015. Structures for structural recursion. In *ICFP*. https://doi.org/10.1145/2858949.2784762

Jean-Yves Girard. 1987. Linear Logic. *Theor. Comput. Sci.* 50 (1987), 1–102. https://doi.org/10.1016/0304-3975(87)90045-4

Robert Harper. 1992. Constructing Type Systems over an Operational Semantics. *J. Symb. Comput.* 14, 1 (1992). https://doi.org/10.1016/0747-7171(92)90026-Z

Hugo Herbelin. 2011. C'est maintenant qu'on calcule, au cœur de la dualité. (2011). http://pauillac.inria.fr/~herbelin/habilitation/ Thèse d'habilitation à diriger les recherches (French).

Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. 2013. Logical Relations and Parametricity – A Reynolds Programme for Category Theory and Programming Languages. In *WACT*. https://doi.org/10.1016/j.entcs.2014.02.008

Danko Ilik. 2013. Continuation-passing style models complete for intuitionistic logic. *Ann. Pure Appl. Logic* 164, 6 (2013). https://doi.org/10.1016/j.apal.2012.05.003

Jean-Louis Krivine. 2014. On the Structure of Classical Realizability Models of ZF. In *TYPES*.

Jean-Louis Krivine. 1985. Un interpreéteur du lambda-calcul. (1985). https://www.irif.fr/~krivine/articles/interprt.pdf

Jean-Louis Krivine. 2007. A call-by-name lambda-calculus machine. *Higher-Order and Symbolic Computation* (2007). https://doi.org/10.1007/s10990-007-9018-9

Jean-Louis Krivine. 2008. Structures de réalisabilité, RAM et ultrafiltre sur N. (Sept. 2008). https://hal.archives-ouvertes.fr/hal-00321410 34 p.

Jean-Louis Krivine. 2009. Realizability in classical logic. *Panoramas et synthèses* 27 (2009), 197–229. https://hal.archives-ouvertes.fr/hal-00154500

Rodolphe Lepigre. 2016. A Classical Realizability Model for a Semantical Value Restriction. In *ESOP*.

Pierre Letouzey. 2008. Extraction in Coq: An Overview. In *CiE*. https://doi.org/10.1007/978-3-540-69407-6_39

Paul Blain Levy. 2004. *Call-By-Push-Value: A Functional/Imperative Synthesis.* Semantics Structures in Computation, Vol. 2. Springer.

Sam Lindley and Ian Stark. 2005. Reducibility and ⊤⊤-lifting for computation types. In *TLCA*. https://doi.org/10.1007/11417170_20

Alexandre Miquel. 2011. A Survey of Classical Realizability. In *TLCA*. https://doi.org/10.1007/978-3-642-21691-6_1

Alexandre Miquel. 2018. Implicative algebras: a new foundation for realizability and forcing. *arXiv e-prints*, Article arXiv:1802.00528 (Feb 2018), arXiv:1802.00528 pages. arXiv:math.LO/1802.00528

Guillaume Munch-Maccagnoni. 2009. Focalisation and Classical Realisability (version with appendices). In *CSL*. http://hal.inria.fr/inria-00409793

Guillaume Munch-Maccagnoni. 2012. λ-calcul, machines et orthogonalité. (2012). http://guillaume.munch.name/files/gdt-logique_24-10-11.pdf Lecture notes from the Groupe de Travail de Logique.

Guillaume Munch-Maccagnoni. 2013. *Syntax and Models of a Non-Associative Composition of Programs and Proofs.* Ph.D. Dissertation. Université Paris Diderot – Paris VII.

G. Munch-Maccagnoni and G. Scherer. 2015. Polarised Intermediate Representation of Lambda Calculus with Sums. In *LICS*. https://doi.org/10.1109/LICS.2015.22

Paulo Oliva and Thomas Streicher. 2008. On Krivine's Realizability Interpretation of Classical Second-Order Arithmetic. *Fundam. Inform.* 84, 2 (2008).

Andrew M. Pitts. 2000. Operational Semantics and Program Equivalence. In *Applied Semantics, International Summer School, APPSEM*. https://doi.org/10.1007/3-540-45699-6_8

Andrew M Pitts and Ian DB Stark. 1998. Operational reasoning for functions with local state. *Higher order operational techniques in semantics* (1998).

Lionel Rieg. 2014. *On Forcing and Classical Realizability. (Forcing et réalisabilité classique)*. Ph.D. Dissertation. École normale supérieure de Lyon, France. https://tel.archives-ouvertes.fr/tel-01061442

Gabriel Scherer. 2016. *Which types have a unique inhabitant? Focusing on pure program equivalence*. Ph.D. Dissertation. Paris Diderot University, France. https://tel.archives-ouvertes.fr/tel-01309712

William W Tait. 1967. Intensional interpretations of functionals of finite type I. *The journal of symbolic logic* 32, 2 (1967). https://doi.org/10.2307/2271658

The Coq Development Team. 2018. *The Coq proof assistant reference manual*. http://coq.inria.fr Version 8.7.

Amin Timany, Léo Stefanesco, Morten Krogh-Jespersen, and Lars Birkedal. 2018. A logical relation for monadic encapsulation of state: proving contextual equivalences in the presence of runST. In *POPL*. https://doi.org/10.1145/3158152

Aaron Joseph Turon, Jacob Thamsborg, Amal Ahmed, Lars Birkedal, and Derek Dreyer. 2013. Logical relations for fine-grained concurrency. In *POPL*. https://doi.org/10.1145/2480359.2429111

Noam Zeilberger. 2009. *The Logical Basis of Evaluation Order and Pattern-Matching*. Ph.D. Dissertation. Carnegie Mellon University.

Noam Zeilberger. 2013. Polarity in Proof Theory and Programming. (August 2013). http://noamz.org/talks/logpolpro.pdf Lecture Notes for the Summer School on Linear Logic and Geometry of Interaction in Torino, Italy.