Realizability with constants

Emmanuel Beffara
Université Paris 7 — Équipe Preuves, Programmes, Systèmes

March 31, 2003

1 Introduction

The proof/program correspondence has been long known to provide an interpretation of logical rules as elementary programming constructs. From λ-calculus as a formalization of cut-elimination in intuitionistic logic, the theory extended to classical logic by the introduction of global control operators [3, 6]. Extensive work has been done studying the proofs with calculus as an underlying tool, but reversing the process by studying the calculus for itself with logic as a tool is an equally interesting work.

In an attempt to move from the logical framework towards the world of actual programming languages, we extend Krivine's classical realizability [4, 5] to a calculus with some constants and primitives and a call-with-current-continuation construct similar to Felleisen's C operator. We show that this theory extends smoothly by associating an appropriate truth value to concrete types. As a consequence, all results from realizability still hold in this new context; this is especially interesting in the case of specification theorems [2, 1] because they hold a significant meaning from the programmer's point of view.

2 The calculus

The calculus we use in this presentation is built as an extension of $\lambda\kappa$-calculus with integer constants. We first define the terms and the evaluation process, then we explain the typing system over it.

2.1 Terms and evaluation

Define inductively the set Λ of terms and the set Π of evaluation contexts as

constants	$n ::= 0 \mid 1 \mid 2 \mid \cdots$
terms	$t ::= x \mid t\,t \mid \lambda x.t \mid \kappa x.t \mid k_\gamma \mid n \mid \mathbf{succ}\,t \mid \mathbf{iter}(t,t,t)$
contexts	$\gamma ::= [\,] \mid \gamma\,t \mid \mathbf{succ}\,\gamma \mid \mathbf{iter}(\gamma,t,t)$
executables	$e ::= \gamma[\![t]\!]$

Our operational semantics is a straightforward call-by-name evaluation. We define it as a reduction relation over executables. The rules for core λ-calculus and control are

$$\gamma[\![t\,u]\!] \succ \gamma\big[[\![t]\!]u\big] \qquad\qquad \gamma[\![\kappa x.t]\!] \succ \gamma\big[\![t[k_\gamma/x]]\!\big]$$
$$\gamma\big[[\![\lambda x.t]\!]u\big] \succ \gamma\big[\![t[u/x]]\!\big] \qquad\qquad \gamma'\big[[\![k_\gamma]\!]t\big] \succ \gamma[\![t]\!]$$

This defines a relation over executables, since contexts are stable under composition. The rules on the left are simply a decomposition of β-reduction, while those on the right define continuation saving and restoring. The binders λx and κx are dual in the sense that one substitutes terms while the other substitutes continuations. The evaluation rules for constants and the primitive **succ** are defined as

$$\gamma[\![\mathbf{succ}\,t]\!] \succ \gamma\big[\mathbf{succ}[\![t]\!]\big]$$
$$\gamma\big[\mathbf{succ}[\![n]\!]\big] \succ \gamma[\![n+1]\!]$$

where $n+1$ is the integer constant (and not a composed term). The rules for iteration are defined as

$$\gamma[\![\mathbf{iter}(t,u,v)]\!] \succ \gamma\big[\mathbf{iter}([\![t]\!],u,v)\big]$$
$$\gamma\big[\mathbf{iter}([\![0]\!],u,v)\big] \succ \gamma[\![v]\!]$$
$$\gamma\big[\mathbf{iter}([\![n+1]\!],u,v)\big] \succ \gamma\big[[\![u]\!]\,\mathbf{iter}(n,u,v)\big]$$

So the computation uses a call-by-name strategy, except that **iter** evaluates its integer argument completely before actually iterating. This is sort of arbitrary, as one could also imagine a rule to reduce terms like $\mathbf{iter}(\mathbf{succ}\, t, u, v)$, but this would lead to a non-deterministic reduction relation, which is undue complication for the point of this work.

One could want to introduce other primitive operators, such as a comparison to zero or a predecessor. All of this can be done easily on the model of the primitives we just defined, but we restrict ourselves to these two constructs as they are representative examples for what we do later on. Moreover, they are already sufficient to express all primitive recursive functions.

2.2 Typing

The type system is nothing against intuition. We extend second order propositional calculus with a constant truth value Int that represents the integer constants. Formally, the typing rules for core $\lambda\kappa$-calculus are

$$\frac{}{\Gamma, x : A \vdash x : A}\,[\text{axiom}] \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\,[\to\text{i}] \qquad \frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B}\,[\to\text{e}]$$

$$\frac{\Gamma \vdash t : A \qquad X \notin FV(\Gamma)}{\Gamma \vdash t : \forall X\, A}\,[\forall\text{i}] \qquad \frac{\Gamma \vdash t : \forall X\, A}{\Gamma \vdash t : A[B/X]}\,[\forall\text{e}]$$

$$\frac{\Gamma, x : A \to B \vdash t : A}{\Gamma \vdash \kappa x.t : A}\,[\text{Peirce}]$$

The first five rules give a standard presentation, known as natural deduction, for second order propositional intuitionistic logic. Alongside with the sixth rule, known as Peirce's law, we get one possible natural deduction presentation of second order propositional classical logic. Apart from this logical framework, we type constants and primitives as

$$\frac{}{\vdash n : \text{Int}} \qquad \frac{\Gamma \vdash t : \text{Int}}{\Gamma \vdash \mathbf{succ}\, t : \text{Int}} \qquad \frac{\Gamma \vdash t : \text{Int} \quad \Gamma \vdash u : A \to A \quad \Gamma \vdash v : A}{\Gamma \vdash \mathbf{iter}(t, u, v) : A}$$

Further on we will use positive and negative literals as types, naturally defining the absurd as $\bot := \forall X\, X$ and the negation as $\neg A := A \to \bot$. All the types we consider later on live actually in a fragment of this system that correspond to simple types augmented with the constants $\bot$ and Int, so types are implicitly considered universally quantified in every variable.

3 Realizability

In pure λ- and $\lambda\kappa$-calculus, realizability builds models of propositional logic by associating truth values in the form of subsets of Π to each closed formula. Let $⫫ \subseteq \Lambda \times \Pi$ be a set of executables closed by anti-reduction, i.e. if $e \succ e'$ and $e' \in ⫫$ then $e \in ⫫$. The executables in this set are called the *observables*. In the logical setting, $⫫$ is said to realize the absurd, while sets of continuations realize hypotheses and sets of terms realize conclusions.

3.1 The logical fragment

From $⫫$ we deduce a notion of *orthogonality* between terms and continuations: a set X of terms is said to be orthogonal to a set Z of continuations (which we can write $X ⫫ Z$) if for any term $t \in X$ and any continuation $\gamma \in Z$ we have $\gamma[\![t]\!] \in ⫫$. The orthogonal of a given set of terms or continuations is then the largest set orthogonal to it, i.e.

$$X^{⫫} = \{\gamma \mid X ⫫ \gamma\} \qquad \text{and} \qquad Z^{⫫} = \{t \mid t ⫫ Z\}$$

(note that we use the same notation in both cases even if the operations are dual).

Let e be a function from propositional variables into the powerset $\mathcal{P}(\Pi)$, i.e. the attribution of a truth value to each propositional variable. The truth value $[A]_e$ associated to a given type A in the valuation e is defined inductively; for the logical fragment, define

$$[X]_e = e(X) \qquad [A \to B]_e = \left\{\gamma[[\,]\, t] \;\middle|\; t \in [A]_e^{⫫}, \gamma \in [B]_e\right\} \qquad [\forall X\, A]_e = \textstyle\bigcup_{\mathcal{Z} \subseteq \Pi} [A]_{e[\mathcal{Z}/X]}$$

where $e[\mathcal{Z}/X]$ is the environment e where value $\mathcal{Z}$ has been assigned to X. The definition is actually rather intuitive and deduced from the typing and reduction rules: an acceptable context for terms of type $A \to B$ mainly applies a term acceptable for type A to its argument in a context acceptable for type B. Similarly, a context acceptable for terms of type $\forall X\, A$ is a context that is acceptable for any particular $A[B/X]$.

3.2 The programming fragment

The rules above inductively define the valuation of any type except for the constant Int. The way we attribute a value to this type is actually fairly independent from the logical fragment, and this modularity is interesting because it shows how the method can be transposed to a wide range of cases.

There are obviously constraints on the value we can give to [Int], as it is guided by type inference rules. The rule for **iter** implies that if, for any valuation of A, γ is acceptable as a context for A and u and v are acceptable terms for type $A \to A$ and A respectively, then $\gamma[\mathbf{iter}([\,], u, v)]$ must be an acceptable context for integers. We thus define

$$[\mathrm{Int}]_0 := \{\gamma[\mathbf{iter}([\,], u, v)] \mid \exists \mathcal{Z} \subseteq \Pi, \gamma \in \mathcal{Z}, u \in [\mathcal{Z} \to \mathcal{Z}]^{\perp\!\!\!\perp}, v \in \mathcal{Z}^{\perp\!\!\!\perp}\}$$

Now the other rule that implies Int in the premises is the one for **succ**, which implies that if γ is in [Int], then $\gamma[\mathbf{succ}\,[\,]]$ must also be in [Int]. Finally, the value of [Int] must ensure that integer constants realize their type. This is formalized as the inequation

$$[\mathrm{Int}] \;\subseteq\; \big\{\gamma[\mathbf{succ}\,[\,]] \bigm| \gamma \in [\mathrm{Int}]\big\} \cup [\mathrm{Int}]_0 \;\subseteq\; \{0, 1, 2, \ldots\}^{\perp\!\!\!\perp}$$

Therefore valid valuations of Int are subsets of Π that satisfy the previous equation. An interesting aspect is that the least fixed point, namely $[\mathrm{Int}]_0$ saturated with **succ**, imposes that all integers are hidden inside iterators, i.e. that in a sense they are not actual first-class (observable) objects, but only serve as an internal component for Church integers.

If we were to introduce other primitives, we would have to proceed the same way: the rules that eliminate Int, in the style of **iter**, define a set of contexts that must be present, while rules that have Int both as premise and conclusion define equations over Int. On the whole, they define a monotonic operator over parts of Π that depends on $\perp\!\!\!\perp$ and whose least fixed point is what we need as a valuation for our type constant.

3.3 Adequacy

We have now defined, for a particular choice of $\perp\!\!\!\perp$ and [Int], a valuation for all types. In a given valuation e, we call *interpretation* of A the set $|A|_e = [A]_e^{\perp\!\!\!\perp}$ of terms orthogonal to $[A]_e$, and we say that a term t *realizes* a type A, which we write $t \vDash A$, if t is in the interpretation of A.

This $\vDash$ is a semantic relation between terms and types, while the typing provided a syntactic relation. A crucial result is that for any choice of $\perp\!\!\!\perp$ an [Int], the typing relation is actually a subset of the realization relation:

Theorem 1 (adequacy). *Let $x_1 : A_1, \ldots, x_n : A_n \vdash t : B$ be a derivable typing judgment and let $\perp\!\!\!\perp$ be a set of observables. For any appropriate value of* [Int] *and any valuation $e : Var \to \mathcal{P}(\Pi)$, for any family $t_1, \ldots, t_n$ of terms such that $t_i \in |A_i|_e$ for each i and any context $\gamma \in [B]_e$, the executable $\gamma[\![t[t_1/x_1, \ldots, t_n/x_n]]\!]$ is in $\perp\!\!\!\perp$.*

The proof of this theorem is not really difficult as it derives rather directly from definitions of truth values. We do not include it here; for the logical fragment, the interested reader can find it in related papers [1, 2].

For closed formulas we can write $[A]$ instead of $[A]_e$ since the value is environment-independent (more generally, $[A]_e$ depends only on the values e takes on variables free in A). In this case, the theorem reduces into the following adequacy lemma:

Proposition 2 (closed adequacy). *For any term t and any type A, if $\vdash t : A$ is derivable then $t \vDash A$ holds for any value of $\perp\!\!\!\perp$ and* [Int].

The fact that this result is valid for any choice of $\perp\!\!\!\perp$ and [Int] gives it a significant power. By tuning this choice for particular cases, we derive easy proofs for specification results. The following consistency property is an interesting example:

Theorem 3. *For any term t of type* Int*, there is a constant n such that $[\![t]\!]$ evaluates into $[\![n]\!]$.*

Proof. Define $\perp\!\!\!\perp$ as the closure by anti-reduction of $\{[\![n]\!] \mid n \in \omega\}$ and [Int] as the least valid subset of Π that contains the context $[\,]$ (that is $[\mathrm{Int}]_0 \cup \{[\,]\}$ saturated by **succ**). From the adequacy lemma we know that t realizes Int, so $[\![t]\!]$ is in $\perp\!\!\!\perp$, which means that it reduces into some $[\![n]\!]$. ☐

Extending realizability to this calculus with individuals gives us, for free, many of the results that this theory derives. With respect to computation, this notably contains a range of specification theorems that describe with precision the behavior inherent to types. As explained in related works [2, 1], this can be a way to synthesize safe and grounded control forms; as shown above, this can be a way to prove valid termination of typed terms.

The example that we exposed here suggests a general pattern for dealing with constants. New primitives over integers would be trivially appended, as well as other types like booleans. The same method could even be extended to parametrized types such as lists, or more generally to inductive types. At this point, inductive types are a no more than a direction for future work; significant literature exists in the intuitionistic setting, it would be interesting to see how it transposes in the context of classical realizability.

References

[1] Emmanuel Beffara and Vincent Danos. Disjunctive normal forms and local exceptions. Submitted to ICFP, 2003.

[2] Vincent Danos and Jean-Louis Krivine. Disjunctive tautologies as synchronisation schemes. In Peter Clote and Helmut Schwichtenberg, editors, *Proceedings of CSL'00*, number 1862 in Lecture Notes in Computer Science, pages 292–301, Fischbachau, 2000. Springer Verlag.

[3] Timothy G. Griffin. A formulae-as-types notion of control. In *17th Symposium on Principles of Programming Languages*, pages 47–58. ACM, January 1990.

[4] Jean-Louis Krivine. Typed lambda-calculus in classical Zermelo-Frænkel set theory. *Archive in Mathematical Logic*, 40(3):189–205, 2001.

[5] Jean-Louis Krivine. Dependent choice, 'quote' and the clock. *Theoretical Computer Science*, to appear.

[6] Michel Parigot. $\lambda\mu$-calculus: an algorithmic interpretation of classical natural deduction. In *Proceedings of Conference on Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. 1992.