

Two talks about specification and objects

Agay Spring School, March 29, 2002

Jean-Louis Krivine

PPS Group, University Paris VII, C.N.R.S.

e-mail krivine@pps.jussieu.fr

In these talks, which will be developed in forthcoming papers, I give two cases of a very general and interesting problem which arises naturally from the Curry-Howard correspondence, and which I call the *specification problem*. Indeed, if you take seriously the correspondence : theorem \Rightarrow specification, then you ask the following : given a mathematical theorem Θ , what are the common features of *all* the programs which are associated with *all* the proofs of Θ ? If you can answer this question, you have found the specification associated with Θ .

The naive answer to this problem is : the specification associated with a theorem is given by its very text. In fact, this is true for any arithmetical theorem of the form $(\forall x \in \mathbb{N})(\exists y \in \mathbb{N})F(x, y)$ where F is a recursive predicate ; like : " There are infinitely many prime numbers ". But this naive idea is completely false in practically every other case, even the simplest ones. Take, for example, the excluded middle which corresponds to control instructions ; this specification was only found rather recently, about ten years ago, which means it was not so trivial !

In order to study this problem, there is a powerful tool, which is often very helpful : the method of *realizability in classical logic*. The realizability in intuitionistic logic was used with great success, particularly by Tait and by Girard, to prove strong normalisation theorems. Some substantial adaptations are needed in order to use it in the context of classical logic, and they are (briefly) explained in these lectures.

In the first lecture, I consider the specification problem for arithmetical theorems in prenex form like before, but with an arbitrary prefix ;

for example, $(\forall x \in \mathbb{N})(\exists y \in \mathbb{N})(\forall z \in \mathbb{N})F(x, y, z)$ where F is again a recursive predicate. The famous Roth's theorem, about rational approximations of algebraic numbers, is of this form and it is known to be non constructive.

The second lecture deals with the trivial valid formula : $\exists x(Px \rightarrow \forall y Py)$, which is sometimes called the "drinker's theorem".

It is interesting to observe that, in both cases, object oriented programming appears in a natural way. Moreover, the second example seems to give a good theoretical basis for this programming style.

Arithmetical theorems call-by-value, objects

Jean-Louis Krivine

PPS Group, University Paris 7, CNRS

krivine@pps.jussieu.fr

Two games for a formula

Let $\Phi \equiv \exists x_1 \forall y_1 \dots \exists x_k \forall y_k F(x_1, y_1, \dots, x_k, y_k)$ (F is quantifier-free).

Consider these games between *the player* \exists and *the opponent* \forall :

Game 1. \exists plays $n_1 \in \mathbb{N}$, \forall plays $p_1 \in \mathbb{N}$, ..., \exists plays n_k , \forall plays p_k .
 \exists **wins** iff $F(n_1, p_1, \dots, n_k, p_k)$.

Game 2. No change for \forall , but a much better situation for \exists :

A *position* of the game is an integer sequence $n_1 p_1 \dots n_i p_i$ ($0 \leq i < k$).

\exists chooses first an already reached position $n_1 p_1 \dots n_i p_i$ (possibly $i = 0$) then an integer n_{i+1} . \forall chooses an integer p_{i+1} .

If $i + 1 < k$, they have reached the position $n_1 p_1 \dots n_{i+1} p_{i+1}$. Go on.

If $i + 1 = k$: if $F(n_1, p_1, \dots, n_k, p_k)$, then \exists **won**. Else go on.

Thus \forall **wins** iff the game lasts infinitely long.

Truth of arithmetical formulas

It is trivial that $\mathbb{N} \models \Phi$ iff the player \exists has a winning strategy for game 1. **The same is true for game 2.** But the difference is that, in this case, we can effectively (and easily) describe such a winning strategy. Moreover this strategy does not depend on Φ , but only on the number k of quantifiers.

A universal strategy

The player \exists uses an effective enumeration of \mathbb{N}^k . When he comes to the k -uple $n_1 \dots n_k$, he chooses the longest already reached position of the form $n_1 p_1 \dots n_i p_i$. Then he successively plays n_{i+1}, \dots, n_k regardless of the choices of \forall . Then he takes the next k -uple.

If this play is infinite, we get k functions $f_i(x_1, \dots, x_i)$ such that

$$\mathbb{N} \models \forall x_1 \dots \forall x_k \neg F(x_1, f_1 x_1, x_2, f_2 x_1 x_2, \dots, x_k, f_k x_1 \dots x_k)$$

which is the Skolem form of $\neg\Phi$. Thus $\mathbb{N} \models \neg\Phi$.

Truth of arithmetical formulas (cont.)

Conversely, if $\mathbb{N} \models \neg\Phi$, there exists k functions $f_i(x_1, \dots, x_i)$ such that the Skolem form of $\neg\Phi$ is satisfied.

Of course, they provide a winning strategy to the opponent \forall . QED

On the contrary, in the case of game 1, there is no such universal strategy. Indeed, for a given true formula Φ , it is in general impossible to describe effectively a winning strategy for \exists , even in the case $k = 1$.

All this is pure model theory : it is simply a way to define the satisfaction of a prenex formula in a denumerable model.

Specification of arithmetical theorems

Let us write every quantifier-free arithmetical formula F in the form $\phi(x_1, y_1, \dots, x_k, y_k) = \psi(x_1, y_1, \dots, x_k, y_k)$ where ϕ, ψ are terms built with primitive recursive functions (or even simply with $0, s, +, \times$).

Assume that $\Phi \equiv \exists x_1 \forall y_1 \dots \exists x_k \forall y_k F(x_1, y_1, \dots, x_k, y_k)$ is provable in classical second order arithmetic. Then a natural problem is : not to extract **programs** from the **proofs** of Φ (which is trivial) but to extract a **specification** from the **formula** Φ itself.

In other words : **What is the common behaviour of the programs associated with all the proofs of Φ ?**

I call this question *the specification problem* for the theorem Φ .

50 years ago, in [3,4], G. Kreisel raised a partial form of this problem :

What is the constructive interpretation of such arithmetical theorems Φ ?

He answered by means of his *no-counterexample interpretation*.

Today, our answer to the specification problem will be given by the

Theorem. *The formula Φ corresponds to the following specification : an interactive program which can stand in for the player \exists in the second game associated with Φ and always win.*

The existence of such a program does not seem extraordinary, since we saw that there is a very simple universal winning strategy. But the strategies given by the various proofs of Φ will be typable λ -terms of type Φ . Therefore, we can use them as modules inside a *proved* main program.

The proof of this theorem is interesting, because two essential features of imperative programming languages appear in it, in a natural way :

- *call-by-value* for a given data type (here the type of integers)
- *object oriented programming*, at least for one important aspect which is *dynamic binding*.

We now define *realisability in classical second order logic* which is the essential tool for this proof.

The λ_c -calculus

Λ (resp. Λ_0) is the set of arbitrary (resp. closed) λ_c -terms.

Π is the set of *stacks*. They are built following these rules :

1. Any variable x , and the constant cc are λ_c -terms.
2. If t, u are λ_c -terms and x is a variable, then $(t)u$ and $\lambda x t$ are λ_c -terms.
3. If π is a stack, the constant k_π is a λ_c -term (called a *continuation*).

A stack is a sequence $\pi = t_1. \dots .t_n.\rho$ of closed λ_c -terms t_i ended with a *stack constant* ρ (the *bottom* of the stack) ;

$t.\pi$ denotes the stack obtained by *pushing* t on the *top* of π .

A *process* is a " product " : $t \star \pi$ ($t \in \Lambda_0, \pi \in \Pi$). It can be performed, a λ_c -term alone cannot.

t is called the *head* of the process $t \star \pi$; at each time, the head is the part of the process which is executed.

Execution of processes

Let $\pi, \pi' \in \Pi$ and $t, u \in \Lambda_0$:

$$\begin{array}{ll} tu \star \pi \succ t \star u.\pi & \text{(push)} \\ \lambda x t \star u.\pi \succ t[u/x] \star \pi & \text{(pop)} \\ \mathbf{CC} \star t.\pi \succ t \star k_\pi.\pi & \text{(store the stack)} \\ k_\pi \star t.\pi' \succ t \star \pi & \text{(restore the stack)} \end{array}$$

Now, let \perp be a fixed *cc-saturated* set of processes, i.e. :

$$t \star \pi \in \perp, t' \star \pi' \succ t \star \pi \Rightarrow t' \star \pi' \in \perp$$

A *truth value* is a subset of Λ_0 of the form $P \rightarrow \perp$ for any $P \subset \Pi$.

$$P \rightarrow \perp = \{t \in \Lambda_0 ; (\forall \pi \in P) t \star \pi \in \perp\}$$

The set of truth values is denoted by \mathfrak{R}_\perp or simply \mathfrak{R} .

The truth value of a formula, defined below, will be the set of λ_c -terms which realize this formula.

Typing in classical 2nd order logic

The only logical symbols are \rightarrow , \forall and function symbols on individuals.
 \perp is defined as $\forall X X$; $\exists x F[x]$ as $\forall X \{ \forall x (F[x] \rightarrow X) \rightarrow X \}$; etc.

Let Γ denote $x_1 : A_1, \dots, x_n : A_n$ (*a context*). Typing rules are :

1. $\Gamma \vdash x_i : A_i$ ($1 \leq i \leq n$)
2. $\Gamma \vdash t : A \rightarrow B, \Gamma \vdash u : A \Rightarrow \Gamma \vdash tu : B.$
3. $\Gamma, x : A \vdash t : B \Rightarrow \Gamma \vdash \lambda x t : A \rightarrow B.$
4. $\Gamma \vdash t : (A \rightarrow B) \rightarrow A \Rightarrow \Gamma \vdash \text{cc } t : A.$
5. $\Gamma \vdash t : A \Rightarrow \Gamma \vdash t : \forall x A$ (resp. $\forall X A$) if x (resp. X) is not free in Γ .
6. $\Gamma \vdash t : \forall x A \Rightarrow \Gamma \vdash t : A[\tau/x]$ for every term τ .
7. $\Gamma \vdash t : \forall X A \Rightarrow \Gamma \vdash t : A[\Phi(x_1, \dots, x_n)/Xx_1 \dots x_n]$ for every formula Φ .

This is the comprehension scheme for second order logic.

Realizability

A *model* \mathcal{M} is a set M of individuals, together with an interpretation $f_{\mathcal{M}} : M^k \rightarrow M$ of each k -ary function symbol f .

The domain of variation of k -ary 2nd order variables is \mathfrak{R}^{M^k} where \mathfrak{R} is the set of truth values.

Let A be a closed 2nd order formula with parameters in M and \mathfrak{R}^{M^k} .

Its truth value, defined below, is $|A| = \|A\| \rightarrow \perp$ with $\|A\| \subset \Pi$.

We say that $t \Vdash A$ (t *realizes* A) if $t \in |A|$ i.e. $(\forall \pi \in \|A\|) t \star \pi \in \perp$.

The definition is by induction on A . If A is atomic, i.e. $R(a_1, \dots, a_k)$ with $a_i \in M$ and $R \in \mathfrak{R}^{M^k}$ the definition is evident.

$$\|A \rightarrow B\| = \{t.\pi ; t \Vdash A, \pi \in \|B\|\} ; \quad \|\forall x A\| = \bigcup_{a \in M} \|A[a/x]\|$$

$$\|\forall X A\| = \bigcup \{\|A[\Psi/X]\| ; \Psi \in \mathfrak{R}^{M^k}\}$$

The general specification problem

Given a provable 2nd order formula Φ , what is the common behaviour of the λ_c -terms t such that $\vdash t : \Phi$? In other words :

what is the *specification* associated with the given theorem Φ ?

This a very interesting but difficult problem. Realizability is a valuable tool because it is compatible with classical 2nd order deduction :

Adequation lemma.

*If $x_1 : \Phi_1, \dots, x_n : \Phi_n \vdash t : \Phi$ and if $t_i \Vdash \Phi_i$ ($1 \leq i \leq n$)
then $t[t_1/x_1, \dots, t_n/x_n] \Vdash \Phi$.*

Thus, we study the behaviour of the λ_c -terms which *realize* Φ .

We now use this framework in order to solve the specification problem for an arithmetical theorem in prenex form :

$$\Phi \equiv \exists x_1 \forall y_1 \dots \exists x_k \forall y_k [\phi(x_1, y_1, \dots, x_k, y_k) = 0]$$

The model (when $k = 1$)

Of course, we take \mathbb{N} as the set of individuals, with standard operations. In such models, second order logic (i.e. comprehension axiom) is realized. Unfortunately, and contrary to intuition, the formula $\neg\forall x \text{Int}(x)$, that is *the negation of the induction axiom* is also realized !

Therefore, we must consider the formula Φ^{Int} with quantifiers restricted to the formula $\text{Int}(x) \equiv \forall X[\forall y(Xy \rightarrow Xsy), X0 \rightarrow Xx]$.

If $k = 1$, we get $\Phi \equiv \exists x\forall y[\phi(x, y) = 0]$ and therefore

$$\Phi^{\text{Int}} \equiv \forall x[\text{Int}(x), \forall y(\text{Int}(y) \rightarrow \phi(x, y) = 0) \rightarrow \perp] \rightarrow \perp.$$

In order to define \perp , we add to λ_c -calculus the following constants : κ_{np} ($n, p \in \mathbb{N}$) and κ which is an *input instruction*.

Thus the λ_c -terms become *interactive programs*.

The model (cont.)

The rule of reduction for κ is :

$$\kappa \star s^n 0. \xi. \pi \rightsquigarrow \xi \star s^p 0. \kappa_{np}. \pi'$$

$n, p \in \mathbb{N}$, $\xi \in \Lambda_0$, $\pi, \pi' \in \Pi$ are arbitrary ;

s is a fixed λ -term for the successor in Church integers.

Meaning : the program proposes $n \in \mathbb{N}$, the opponent plays $p \in \mathbb{N}$, and the execution goes on ; κ_{np} keeps a trace of the reached position np .

We define \perp as the set of processes *all* reductions of which end up into $\kappa_{np} \star \pi$ *with* $\phi(n, p) = 0$.

The ordered pair $(s^n 0, \xi)$ is an *object* made up with a *data* n and a *method* ξ . This method uses the data p given by the opponent in order to *update* the value of n .

In the source program, this method has a fixed name ($\kappa.method$), which represents a new piece of code ξ each time it is called (i.e. κ arrives in head position). This an example of *dynamic binding*.

Call-by-name, call-by-value

Define $T = \lambda f \lambda n(n) \lambda g g \circ s.f.0$ (*storage operator* [6]).

Theorem. *If $f.s^n 0 \Vdash X$ then $Tf \Vdash Int(n) \rightarrow X$.*

Proof. Let $\|Pj\| = \{s^{n-j} 0.\pi; \pi \in \|X\|\}$ for $0 \leq j \leq n$ and $\|Pj\| = \emptyset$ for $j > n$. Then $\lambda g g \circ s \Vdash \forall x(Px \rightarrow Psx)$ and $f \Vdash P0$. Thus, if $\nu \Vdash Int(n)$ then $\nu.\lambda g g \circ s.f \Vdash Pn$ which gives $Tf\nu \Vdash X$. QED

Let ν be a λ_c -term which realizes $Int(n)$.

In other words, ν is a program which "behaves like" the integer n .

In the λ_c -term $f\nu$ this data is *called by name* by the program f .

In the λ_c -term $Tf\nu$ the same data is *called by value* by f .

Call-by-value is only defined for data types.

We have got now all the necessary tools in order to show the specification associated with the formula Φ^{Int} .

The main result (for $k = 1$)

Theorem. *If $\theta \Vdash [\exists x \forall y (\phi(x, y) = 0)]^{Int}$ then every reduction of $\theta \star T\kappa.\pi$ following \succcurlyeq ends up into $\kappa_{np} \star \pi'$ with $\phi(n, p) = 0$.*

It follows that a proof of Φ in classical second order arithmetic provides an interactive program which wins against every opponent.

Notice the presence of the storage operator T . It means that the first argument of κ is a *called-by-value integer*.

In other words, it must be computed first.

This is very natural, since the instruction κ introduces the name κ_{np} .

Indeed, after each reply of the opponent, the program provides an *object* $(s^n 0, \xi)$ made up with an integer n (the provisional solution) and an *exception handler* ξ which is used in case of a relevant reply from the opponent.

These are the two arguments of κ which is therefore a *pointer* towards this object.

Proof

We have $\theta \Vdash \forall x[Int(x), \forall y(Int(y) \rightarrow \phi(x, y) = 0) \rightarrow \perp] \rightarrow \perp$.

Thus, by the adequation lemma, it is sufficient to prove that :

$T\kappa \Vdash \forall x[Int(x), \forall y(Int(y) \rightarrow \phi(x, y) = 0) \rightarrow \perp]$.

By the theorem about T , this becomes :

$\kappa.s^n 0 \Vdash \forall y(Int(y) \rightarrow \phi(n, y) = 0) \rightarrow \perp$ for every $n \in \mathbb{N}$.

Let $\xi \Vdash \forall y(Int(y) \rightarrow \phi(n, y) = 0)$, thus $\xi.s^p 0 \in |\phi(n, p) = 0|$ for $p \in \mathbb{N}$.

If $\phi(n, p) = 0$: then $|\phi(n, p) = 0|$ is $|\forall X(X\phi(n, p) \rightarrow X0)|$ or else $|\forall Z(Z \rightarrow Z)|$. But, by definition of \perp , we have also $\kappa_{np} \in \perp$. Therefore :

(*) $\xi.s^p 0.\kappa_{np} \in \perp$ that is $\xi \star s^p 0.\kappa_{np}.\pi \in \perp$ for every stack π .

If $\phi(n, p) \neq 0$: then $|\phi(n, p) = 0|$ is $\top \rightarrow \perp$ where \top is the set of all closed λ_c -terms. Therefore (*) is again true.

Thus, we have $\xi \star s^p 0.\kappa_{np}.\pi \in \perp$ for every $p \in \mathbb{N}$ and $\pi \in \Pi$.

Therefore, by definition of \perp , we get $\kappa \star s^n 0.\xi.\pi \in \perp$.

QED

The general case

The proof is almost the same, just a bit more complicated. We introduce the constants $\kappa_{n_1 p_1 \dots n_i p_i}^i$ for $0 \leq i \leq k$. Their rule of reduction is :

$$\kappa_{n_1 p_1 \dots n_i p_i}^i \star s^{n_i+1} 0. \xi. \pi \succ \xi \star s^{p_i+1} 0. T_{i+1} \kappa_{n_1 p_1 \dots n_{i+1} p_{i+1}}^{i+1} \cdot \pi'$$

with $T_i = T$ for $1 \leq i < k$ and $T_k = I$.

Theorem. *If $\theta \Vdash [\exists x_1 \forall y_1 \dots \exists x_k \forall y_k (\phi(x_1, y_1, \dots, x_k, y_k) = 0)]^{Int}$ then every reduction of $\theta \star T \kappa^0. \pi$ following \succ ends up into $\kappa_{n_1 p_1 \dots n_k p_k}^k \star \pi'$ with $\phi(n_1, p_1, \dots, n_k, p_k) = 0$.*

This means that each proof of Φ^{Int} gives an interactive program, which wins against every opponent, in the second game which gives the satisfaction in \mathbb{N} of the formula $\Phi \equiv \exists x_1 \forall y_1 \dots \exists x_k \forall y_k (\phi(x_1, y_1, \dots, x_k, y_k) = 0)$.

Concluding remarks

The constant $\kappa_{n_1 p_1 \dots n_i p_i}^i$ is a *dynamic name* of a pointer to an object (n_{i+1}, ξ_{i+1}) . This object is made up with the integer proposed by the player \exists (which is, in fact, the program) and a method in order to continue the game.

Of course, the piece of code for this method is itself dynamic.

There is a theory of this programming style (which is nothing else than *object oriented programming*) which uses the λ_c -terms of type

$\forall P \exists x (Px \rightarrow \forall y Py)$ (the so-called "drinker's theorem"). Cf. next lecture.

Note that the drinker's theorem can replace the excluded middle in order to prove a negation $\forall \vec{x} \neg A(\vec{x})$. The prenex arithmetical theorems which we have just considered are of this form, since we translate \exists by $\neg \forall \neg$.

This explains the fact that we obtain programs in object oriented style.

References

1. **Thierry Coquand** *A semantics of evidence for classical arithmetic.*
J. Symbolic Logic 60, pp. 325-337 (1995).
2. **Ulrich Kohlenbach** *On the no-counterexample interpretation.*
J. Symbolic Logic 64, pp. 1491-1511 (1999).
3. **Georg Kreisel** *On the interpretation of non-finitist proofs, part I.*
J. Symbolic Logic 16, pp. 241-267 (1951).
4. **Georg Kreisel** *On the interpretation of non-finitist proofs, part II: Interpretation of number theory, applications.* J. Symbolic Logic 17, pp. 43-58 (1952).
5. **Georg Kreisel** *Mathematical significance of consistency proofs.*
J. Symbolic Logic 23, pp. 155-182 (1958).
6. **Jean-Louis Krivine** *A general storage theorem for integers in call-by-name λ -calculus.* Th. Comp. Sc. 129, pp. 79-94 (1994).

The drinker and the objects

Jean-Louis Krivine

PPS Group, University Paris 7, CNRS

krivine@pps.jussieu.fr

The specification problem (reminder)

Given a provable 2nd order formula Φ , what is the common behaviour of the λ -terms t such that $\vdash t : \Phi$? In other words :

what is the *specification* associated with the given theorem Φ ?

This a very interesting but difficult problem. Realizability is a valuable tool because it is compatible with classical 2nd order deduction :

Adequation lemma.

*If $x_1 : \Phi_1, \dots, x_n : \Phi_n \vdash t : \Phi$ and if $t_i \Vdash \Phi_i$ ($1 \leq i \leq n$)
then $t[t_1/x_1, \dots, t_n/x_n] \Vdash \Phi$.*

Thus, we study the behaviour of the λ_c -terms which *realize* Φ .

We now use this framework in order to solve the specification problem for the so-called *drinker's formula* i.e. :

$$\forall P \exists x (Px \rightarrow \forall y Py)$$

The drinker's term

This formula may be written as :

$$\forall P \forall Y \{ \forall y [(P y \rightarrow \forall x P x) \rightarrow Y] \rightarrow Y \}$$

We shall first study the behaviour of a λ_c -term given by a particular proof (the simplest, of course). We shall see later that *any* term given by any proof behaves in the same way. The λ_c -term in question is :

$$\gamma = \lambda \sigma (\mathbf{CC}) \lambda k (\sigma) \lambda d (\mathbf{CC}) \lambda z (k) (\sigma) z$$

This very short program has a truly remarkable execution : it carries out a *dynamic management of names* (creation and assignment) which seems to be a good theoretical basis for *object oriented programming*.

This λ_c -term is nothing else than a piece of executable machine code without documentation. What we shall do now is precisely *to disassemble* this code, which is very funny hacking.

The proof

$\sigma : \forall y[(Py \rightarrow \forall x Px) \rightarrow Y], k : Y \rightarrow \forall x Px \vdash$

$k \circ \sigma : (Py \rightarrow \forall x Px) \rightarrow Py \ ; \ \lambda d(\text{cc})k \circ \sigma : Py \rightarrow \forall x Px$

Therefore $\sigma : \forall y[(Py \rightarrow \forall x Px) \rightarrow Y] \vdash (\text{cc})\lambda k(\sigma)\lambda d(\text{cc})k \circ \sigma : Y$.

Performing the process $\gamma \star \sigma.\pi$

For $\sigma \in \Lambda_0$ and $\pi \in \Pi$, let $\chi_{\sigma,\pi} = \lambda d(\text{cc})\lambda z(k_\pi)(\sigma)z$. Then we have :

$$\gamma \star \sigma.\pi \succ \sigma \star \chi_{\sigma,\pi}.\pi$$

$$\chi_{\sigma,\pi} \star t.\varpi \succ \sigma \star k_\varpi.\pi$$

Meaning : introduce a *new name* χ and look at the first time it comes in head position, i.e. $\sigma \star \chi.\pi \succ \chi \star t_0[\chi].\varpi_0[\chi]$.

This allows many short-circuits in the remainder of the execution :

$$\chi_{\sigma,\pi} \star t.\varpi \succ t_0[k_\varpi] \star \varpi$$

Indeed $\chi_{\sigma,\pi} \star t.\varpi \succ \sigma \star k_\varpi.\pi \succ k_\varpi \star t_0[k_\varpi].\varpi_0[k_\varpi] \succ t_0[k_\varpi] \star \varpi$

A first approximation

The process $\gamma \star \sigma.\pi$ seems to behave as follows :

$$\gamma \star \sigma.\pi \succ \sigma \star \chi.\pi \succ \chi \star t_0[\chi].\varpi_0[\chi]$$

where χ is a name, created when γ came in head position.

Afterwards, each time χ comes back in head position, it throws away its argument and is replaced with the fixed term $(\text{cc})\lambda\chi t_0[\chi]$.

Indeed $\chi \star t.\varpi \succ t_0[k_\varpi] \star \varpi$ which is equivalent to :

$$\chi \star t.\varpi \succ (\text{cc})\lambda\chi t_0[\chi] \star \varpi.$$

Summary : each execution of γ *creates a name* χ . The first time this name comes in head position, it is *assigned* with its argument $t_0[\chi]$ (in fact by $(\text{cc})\lambda\chi t_0[\chi]$). This is clearly an example of *dynamic binding*.

But all this is only an approximation.

(Un)fortunately, the real thing is more complicated (and more interesting).

The problem of names

The mistake is that we have identified :

- the reduction of $\sigma \star k_{\varpi}.\pi$
- the reduction $\sigma \star \chi.\pi \succ \chi \star t_0[\chi].\varpi_0[\chi]$ in which we replace χ with k_{ϖ} .

But this is valid only if no names are created during these reductions : otherwise, the names created in both reductions are not the same, with catastrophic consequences.

In order to settle this problem, we must introduce serious complications in the reduction rules of processes, and thus in the *implementation*.

But this is more than balanced by the gain in speed at execution, whole pieces of which are short-circuited.

We have to manage changes of names inside objects, at each assignment.

New rules of execution

We add to the λ_c -calculus a constant γ and an infinite set C of new names. We now give the rules of execution, not of a process alone, but of a pair, made up with a *process* and a *list* of

- *creations* : a creation is simply a name $c \in C$;
- and *assignments* : it is a pair $c \# \tau$ (read: c is assigned with τ) with $c \in C$ and $\tau \in \Lambda_0$.

Each name is created and assigned only once (at most).

The assignment for a name always takes place *after* its creation.

At each execution step, we only add new elements *at the end* of the list.

Rule 1

$$\gamma \star \sigma.\pi \succ \sigma \star c.\pi$$

where $c \in C$ does not appear in σ, π .

Add to the list : the creation of c ; no assignment.

Rule 2

$$c \star t.\varpi \succ \tilde{\tau}[k_{\varpi}/c] \star \varpi$$

where τ is given by the assignment of c . In case c is not assigned :
add to the list the assignment $c \# t$ and take $\tau = t$.

Definition of $v \mapsto \tilde{v}$:

For each name $d \neq c$ which is created *between the creation and the assignment*
of c , take simply for \tilde{d} a new name in C ($d \mapsto \tilde{d}$ injective).

Now, to obtain the λ_c -term \tilde{v} , just carry out in v all the substitutions $[\tilde{d}/d]$.

Then, add to the list the creation of these names \tilde{d} .

Add also the assignments $\tilde{d} \# \tilde{v}[k_{\varpi}/c]$

for every name d , the creation and the assignment $(d \# v)$ of which took place
between the creation and the assignment of c .

Take care to order all these creations and assignments in the same way
as the corresponding creations and assignments of the original names d .

Rules 1 and 2 are correct

Let p_0 be a process in which no name in C appear.

Let $(p_0, \Sigma_0), \dots, (p_n, \Sigma_n), \dots$ be the execution following these rules of the pair (p_0, Σ_0) where Σ_0 is the empty sequence.

Then we can define an application $\iota : C \rightarrow \Lambda_0$

(which maps names into closed λ_c -terms)

such that $\iota(p_n)$ is a *cofinal subsequence* of the execution of the process p_0 .

This subsequence skips many steps,

hence an important gain in speed at execution time.

Objects

A name c defines a *class* τ at the moment it is assigned ($c \# \tau$), and an *object* $\tilde{\tau}[k_{\varpi}/c]$ each time it comes in head position with an environment (stack) ϖ (in particular, at the first time, when it is assigned).

A *method* of τ is a name d which is created between the creation and the assignment of c . There are two possibilities :

- d is assigned before c : then the method is the same in each object of the class τ . It is a part of the fixed structure of this class.
- d is assigned after c : then the method will be assigned in various ways in each object of the class τ . By means of these new classes, the structure of the class τ can be enriched arbitrarily.

This gives a way to formalize *inheritance*.

Henkin deduction system

To each formula $F[x]$ (resp. $F[X]$) with only one free variable, we associate a new constant a (resp. A) : the *Henkin witness* of F .

The introduction *rule* for \forall is replaced with the Henkin *axiom* :

$\vdash \chi_F : F[a/x] \rightarrow \forall x F[x]$ for first order

$\vdash \chi_F : F[A/X] \rightarrow \forall X F[X]$ for second order.

This gives a conservative extension of classical second order logic.

χ_F is the name associated with the formula F .

For example, the Henkin witness of the formula $\forall X X$ is \perp .

In this system, there is an *intuitionistic* proof of the drinker's formula :

$\vdash \chi_0 \lambda f f \chi_1 : \forall P \forall Y \{ \forall y [(Py \rightarrow \forall x Px) \rightarrow Y] \rightarrow Y \} \equiv \forall P \Phi[P]$

where χ_0 is the name associated with $\Phi[P]$ and χ_1 with Ax

where A is the Henkin witness of $\Phi[P]$.

Realization of Henkin axioms

Let F be a formula without any witness and

$\chi_1:A_1[a_1] \rightarrow \forall x_1 A_1[x_1], \dots, \chi_n:A_n[a_n] \rightarrow \forall x_n A_n[x_n] \vdash \tau[\chi_1, \dots, \chi_n]:F$

a proof of F in this system. We assume that a_i, \dots, a_n do not appear in $A_i[x_i]$, in other words $A_i[x_i] \equiv A_i[a_1, \dots, a_{i-1}, x_i]$.

Then, the process $\tau[\chi_1, \dots, \chi_n] \star \pi$ must be performed, assuming that the names χ_1, \dots, χ_n are created in this order and not yet assigned.

Indeed, we show easily that $\vdash \gamma\lambda\chi_1 \dots \gamma\lambda\chi_n \tau[\chi_1, \dots, \chi_n] : F$.

Taking $\pi \in \llbracket F \rrbracket$, we have $\gamma\lambda\chi_1 \dots \gamma\lambda\chi_n \tau \star \pi \in \perp$. If this process is performed following rules 1 and 2, we immediately get $\tau \star \pi$.

Example 1

$\chi_0 \lambda f f \chi_1$ and γ are executed in the same way, and both have the type of the drinker's formula.

Example 2. Update

Consider a program like $\chi_0 t[\chi_1 n_0, \chi_1 n_1, x_0, x_1, \dots]$; assume that the names χ_0, χ_1 are created in this order and not yet assigned (we may, for example, put $\gamma\lambda\chi_0\gamma\lambda\chi_1$ as a heading of the process) ; n_0, n_1 are λ_c -terms of integer type which may contain the variables x_i . At execution, χ_0 is assigned with t , then χ_1 is assigned by the term n_i such that $\chi_1 n_i$ comes first in head position, say $\chi_1 n_0$. After that, it does not take care of the others : this means that $\chi_1 n_k$ may now be read as $\chi_1 n_0$. This is valid until χ_0 comes back in head position. Then χ_1 is assigned with a new n_i when it comes back in head position. We see that the value of χ_1 is *updated*, and this update is controlled by χ_0 . It is a way to realize a standard example of object : a memory cell with two methods : reading and writing.

Specification of drinker's formula

So far, we have analysed the behaviour of a particular λ_c -term :

$$\gamma = \lambda\sigma(\mathbf{cc})\lambda k(\sigma)\lambda d(\mathbf{cc})\lambda z(k)(\sigma)z$$

which is the simplest of type $\forall P\exists x[Px \rightarrow \forall y Py]$.

Does any term γ' of this type behave like γ , in some sense ?

The following theorem tells us that γ and γ' are *operationally equivalent*.

I think that much stronger statements are true.

Theorem. *Let γ' be a λ_c -term such that $\vdash \gamma' : \forall P\exists x[Px \rightarrow \forall y Py]$; and $p[x, a_0, \dots, a_n]$ a process with one free variable and some constants. Then $p[\gamma, a_0, \dots, a_n] \succ a_0 \star \pi$ for some stack π*

iff $p[\gamma', a_0, \dots, a_n] \succ a_0 \star \pi'$ for some stack π' .

Moreover, the stacks π and π' have the same length and the same bottom stack constant.