Proof pearl: from concrete to functional unparsing

Jean-François Monin

VERIMAG - Centre Equation 2 avenue de Vignate, F-38610 Gières, France
jean-francois.monin@imag.fr,
WWW home page: http://www-verimag.imag.fr/~monin/

Abstract. We provide a proof that the elegant trick of Olivier Danvy for expressing printf-like functions without dependent types is correct, where formats are encoded by functional expressions in continuation-passing style. Our proof is formalized in the Calculus of Inductive Constructions. We stress a methodological point: when one proves equalities between functions, a common temptation is to introduce a comprehension axiom and then to prove that the considered functions are extensionally equal. Rather than weakening the result (and adding an axiom), we prefer to strenghten the inductive argumentation in order to stick to the intensional equality.

1 Introduction

In [1], Olivier Danvy proposes an elegant trick for expressing printf-like functions and procedures in the ML type system. His idea is to replace the concrete version of the first argument, on which the number and the type of remaining arguments depend, with a higher-order function. In order to avoid questions related to side-effects, let us consider the *sprintf* function, which builds a string from its arguments. The first argument of *sprintf* is a *format*, which specifies the number and the type of the remaining arguments. In practice, notably in the C language, the format is often a string, where occurrences of `%d` (respectively, of `%s`, etc.) specify that an integer (respectively, a string, etc.) should be inserted there. For instance, in ML syntax,

$$\textit{sprintf}\ \texttt{"The \%s is \%d \%s."}\ \texttt{"distance"}\ \texttt{10}\ \texttt{"meters"} \tag{1}$$

would return the string `"The distance is 10 meters."`.

It is more convenient, at least for reasoning purposes, to represent formats using a concrete type such as lists of an appropriate type of directives. For example, the first argument of (1) could be represented by

$$[\textit{Lit}(\texttt{"The "});\ \textit{String};\ \textit{Lit}(\texttt{" is "});\ \textit{Int};\ \textit{String};\ \textit{Lit}(\texttt{"."})]. \tag{2}$$

In a language where dependent types are allowed, it is then a simple exercise to program the desired behavior. In the case of ML, Danvy proposes to represent the format by a functional expression:

$$\textit{lit}\,\texttt{"The "} \circ \textit{str} \circ \textit{lit}\,\texttt{" is "} \circ \textit{sint} \circ \textit{str} \circ \textit{lit}\,\texttt{"."}, \tag{3}$$

where $\circ$ is the sequential composition of functions, and the functions such as `int` and `string` take a continuation on strings, a string, an argument of the appropriate type and return a continuation on strings. More specifically, *str* is defined by $\lambda k\, a\, s.\, k(a\,\hat{}\,s))$, where $\hat{}$ is string catenation and *sint* is defined by $\lambda k\, a\, n.\, k(a\,\hat{}\,\mathit{string\_of\_int}\, n))$. The definition of *lit* is $\lambda s\, k\, a.\, k(a\,\hat{}\,s))$. Reducing these definitions in (3) yields

$$\lambda k\, a\, s_1\, n\, s_2.\, k\,(a\,\hat{}\,\texttt{"The "}\,\hat{}\,s_1\,\hat{}\,\texttt{" is "}\,\hat{}\,\mathit{string\_of\_int}\, n\,\hat{}\,s_2\,\hat{}\,\texttt{"."}) \qquad (4)$$

and we see that applying the following continuation-based version of *sprintf* to a functional format does the job.

$$\mathit{sprintfk} := \lambda f.\, f\,(\lambda s.\, s)\,\texttt{""} \qquad (5)$$

An interesting feature of functional formats is that they are more general than concrete formats given by either a string as in (1), or a list as in (2): concrete formats are bound to a fixed number of data types, whereas functional formats are extensible—they can handle any data type X, provided we are given a function from X to *string*.

If we look at types, we remark that *str* has the type $(\mathit{string} \to \beta) \to \mathit{string} \to \mathit{string} \to \beta$, *sint* has the type $(\mathit{string} \to \alpha) \to \mathit{string} \to \mathit{int} \to \alpha$, hence $\mathit{str} \circ \mathit{sint}$ has the type $(\mathit{string} \to \alpha) \to \mathit{string} \to \mathit{string} \to \mathit{int} \to \alpha$. In general, the type of a functional format has the form $(\mathit{string} \to \alpha) \to \mathit{string} \to X_1 \to \cdots \to X_n \to \alpha$. If two formats f_1 and f_2 are respectively of type

$$(\mathit{string} \to \beta) \to \mathit{string} \to X_1 \to \cdots \to X_n \to \beta \qquad (6)$$

and

$$(\mathit{string} \to \alpha) \to \mathit{string} \to X_{n+1} \to \cdots \to X_{n+p} \to \alpha, \qquad (7)$$

their composition $f_1 \circ f_2$ is of type

$$(\mathit{string} \to \alpha) \to \mathit{string} \to X_1 \to \cdots \to X_n \to X_{n+1} \to \cdots \to X_{n+p} \to \alpha \qquad (8)$$

while the type inference mechanism yields

$$\beta = X_{n+1} \to \cdots \to X_{n+p} \to \alpha. \qquad (9)$$

We provide here a formal proof that Danvy's functional formats are correct representations of usual concrete formats. More precisely, for any concrete format ϕ, we inductively define its functional representation $\mathit{kformat}\,\phi$ and we prove that *sprintf* applied to ϕ yields the same function as *sprintfk* applied to $\mathit{kformat}\,\phi$—these functions are even convertible.

As the result relates dependent types with polymorphic types, we need a logic where these two features are present. Our formalization is carried out in the Calculus of Inductive Constructions [2]. Two proof techniques are illustrated: making the statement of an inductive property on functions more intensional, rather than reasoning on extensional equality; and using type transformers to recover what is performed by type inference in (9). A complete Coq script (V8.0) is available on the web page of the author.

2 Type theory and notation

The fragment of the Calculus of Inductive Constructions to be used here includes a hierarchy of values and types. On the first level we have basic inductive and functional values such as 0 and $\lambda x : nat.\ x$. They inhabit types such as *nat* or $nat \to nat$, which are themselves values at the second level and have the type *Set*. In the sequel α, β, γ, δ range over such types. Polymorphic types are obtained using explicit universal quantification, e.g. $\forall\alpha\ \alpha \to \alpha$. We can also construct type transformers such as $\lambda\alpha.\ nat \to \alpha$, of type $Set \to Set$. The type of *Set* and of $Set \to Set$ is called *Type*. Types can depend on values of any level.

Functions are defined using the following syntax:
Definition $function\_name\ arg_1 \ldots arg_n : type\_of\_the\_result := body.$
where $type\_of\_the\_result$ depends on $arg_i, i \in 1 \ldots n$. In the case of a recursive definition, Definition is replaced with Fixpoint.

We suppose that we are given a type *string* (in *Set*), endowed with a binary operation $\_\hat{}\_$ (catenation) and the empty string denoted by `""`. We don't need an algebraic law for catenation.

3 Concrete formats

Our definitions will be illustrated on a format specified by `"foo %s bar %i"` in C language notation. Assuming two strings *foo* and *bar*, the structured concrete representation that we will use is:

$$\textsf{Definition}\ example := Lit\ foo\ (Str\ (Lit\ bar\ (Int\ Stop))) \qquad (10)$$

where *Str* and *Int* are respectively of type $string \to format \to format$ and $int \to format \to format$; *format* is a dedicated inductive type defined below.

For the sake of generality (functional formats can handle arbitrary printable data) we first introduce a structure for printable data, composed of a carrier X and of a function r_X which appends a printed representation of a value of type X to the right of a given string. This is equivalent to providing a function for converting an inhabitant of X to a string, but turns out to be much more handy.

Record $Printable : Type := mkpr\ \{X : Set;\ r\_X : string \to X \to string\}.$

In Coq, a record is just a tuple and fields are represented by projections. For our example, we suppose that we are given a type *int* for integers and a corresponding function *r_int*. Then we can define *pint* as *mkpr int r_int*, and we have $X\ pint = int$ and $r\_X\ pint = rint$. We use the notation "$a\ \hat{}_P\ x$" for $r\_X\,P\,a\,x$, where P is a *Printable*, a is a string and x is an $X\,P$.

The type of concrete formats is given by:

Type $format := Stop \mid Data$ of $Printable \times format \mid Lit$ of $string \times format.$

In our example, *Int* is defined as *Data pint*. Note that from printable integers, it is easy to add printable lists of integers and so on.

In the sequel, ϕ ranges over *format* and P ranges over *Printable*.

4 A first translation

In this section, we work with a monomorphic version of Danvy's functional formats. This it is not satisfactory, but the proof technique that we want to use is simple to explain. Polymorphic functional formats will be considered in section 5.

The type associated to a format is:

Fixpoint *type_of_fmt* ϕ : *Set* :=
  match ϕ with
  | *Stop* $\Rightarrow$ *string*
  | *Data* $P\ \phi \Rightarrow X P \rightarrow$ *type_of_fmt* ϕ
  | *Lit* $s\ \phi \Rightarrow$ *type_of_fmt* ϕ
  end.

For example, *type_of_fmt example* reduces to *string* $\rightarrow$ *int* $\rightarrow$ *string*.

4.1 Basic version of sprintf with dependent types

We start with a loop which prints on the right of an additional argument.

Fixpoint *r_sprintf* ϕ : *string* $\rightarrow$ *type_of_fmt* ϕ :=
  match ϕ with
  | *Stop* $\Rightarrow \lambda a.\ a$
  | *Data* $P\ \phi \Rightarrow \lambda a\, x.$ *r_sprintf* $\phi\ (a\ \hat{}_P\ x)$
  | *Lit* $s\ \phi \Rightarrow \lambda a.$ *r_sprintf* $\phi\ (a\,\hat{}\,s)$
  end.

The desired function provides the empty string "" as the initial accumulator to the previous function.

Definition *sprintf* ϕ := *r_sprintf* ϕ "".

4.2 Monomorphic functional formats

The following type of Danvy's *sprintfk* is allowed in the Damas-Milner type system, it can then be used in languages of the ML and Haskell family. Though there is no restriction over α, the only form α can take is (*type_of_fmt* ϕ) for some format ϕ. However, the point is that ϕ itself is no longer an argument of *sprintfk*.

Definition *sprintfk*: $\forall\alpha\ ((string \rightarrow string) \rightarrow string \rightarrow \alpha) \rightarrow \alpha := \lambda f.\, f\,(\lambda s.\, s)$ "".

Functional formats are constructed using primitive formats such as *lit*, *str*, *sint*, etc. The two latter are themselves special cases of our *kdata*, which is not admitted in ML, in contrast with *str*, *sint*, etc. However we keep *kdata* here for the sake of generality in the reasoning. In ML examples, we use only instances of *kdata*.

Definition *kid* : $(string \rightarrow string) \rightarrow string \rightarrow string := \lambda k\, a.\ k\, a$.

Definition *kdata* P : $\forall\alpha\ (string \rightarrow \alpha) \rightarrow string \rightarrow X P \rightarrow \alpha$:=
  $\lambda\alpha.\ \lambda k.\ \lambda a\, x.\ k\,(a\ \hat{}_P\ x)$.

Definition *lit* $(x{:}string)$: $\forall\alpha\ (string \rightarrow \alpha) \rightarrow string \rightarrow \alpha := \lambda\alpha.\ \lambda k\, a.\ k\,(a\,\hat{}\,x)$.

4.3 Translation

Here is the general construction of functional formats from concrete formats.

Fixpoint $kformat\ \phi : (string \to string) \to string \to type\_of\_fmt\,\phi :=$
match ϕ with
$\mid Stop \Rightarrow kid$
$\mid Data\ P\ \phi \Rightarrow (kdata\ P\ (type\_of\_fmt\ \phi)) \circ (kformat\ \phi)$
$\mid Lit\ x\ \phi \Rightarrow (lit\ x\ (type\_of\_fmt\ \phi)) \circ (kformat\ \phi)$
end.

For example, *kformat example* is convertible with
$(lit\ foo\ (string \to int \to string)) \circ$
$(str\ (int \to string)) \circ (lit\ bar\ (int \to string)) \circ (sint\ string)$.

4.4 Correctness of sprintfk w.r.t. sprintf

A brutal attempt to prove that $(sprintf\ \phi) = (sprintfk\ (kformat\ \phi))$ holds for all ϕ fails, because the accumulator changes at each recursive call (an induction on ϕ would lead us to to prove something on "" ^ s while the induction hypothesis is on ""). The usual trick is then to replace "" with a variable (let us call it a) which is in the scope of the induction. We first unfold *sprintf* and *sprintfk* in order to work with *r_sprintf* and *kformat*. Then, if we try to prove

$$\forall a\ \ r\_sprintf\,\phi\ a \ = \ kformat\,\phi\ (\lambda s.\ s)\ a \tag{11}$$

by induction on ϕ, we face another problem: how to prove

$$\lambda x.\ r\_sprintf\,\phi\ (a\ \hat{}_P\ x) \ = \ \lambda x.\ kformat\,\phi\ string\ (\lambda s.\ s)\ (a\ \hat{}_P\ x)$$

from the induction hypothesis (11)? This is a typical case where extensionality makes life easier. Adding the following axiom would allow us to finish the proof in a trivial way.

Axiom *extensionality*:
$\forall \alpha\,\beta,\ \forall f g : \alpha \to \beta,\ \ (\forall x{:}\alpha,\ f\,x = g\,x) \to (\lambda x.\,f\,x) = (\lambda x.\ g\,x)$.

But this workaround is not satisfactory. In order to prove the desired (intensional) equality, without any additional axiom, we work with a *still more intensional* statement:

$$\lambda a.\ r\_sprintf\,\phi\ a \ = \ \lambda a.\ kformat\,\phi\ (\lambda s.\ s)\ a \tag{12}$$

or even a η-reduced version of the latter:

$$r\_sprintf\,\phi \ = \ kformat\,\phi\ (\lambda s.\ s). \tag{13}$$

The proof is very short. The key is to observe that $\lambda a\,x.\ k\ (a\ \hat{}_P\ x) = kdata\ P\ \alpha\ k$, and similarly for *lit*. We can then rewrite *r_sprintf* as follows:

Fixpoint $r\_sprintf1\ \ \phi : string \to type\_of\_fmt\ \phi\ :=$

match ϕ with
| $Stop \Rightarrow \lambda a.\ a$
| $Data\ P\ \phi \Rightarrow kdata\ P\ (type\_of\_fmt\ \phi)\ (r\_sprintf1\ \phi)$
| $Lit\ s\ \phi \Rightarrow lit\ s\ (type\_of\_fmt\ \phi)\ (r\_sprintf1\ \phi)$
end.

The following lemma is easily proved by induction on ϕ:

$$\forall \phi\ r\_sprintf1\ \phi = kformat\ \phi\ (\lambda s.\ s). \tag{14}$$

Unfolding definitions and converting *r_sprintf* to *r_sprintf1* provides the desired corollary.

Theorem *sprintf_sprintfk*: $\forall \phi\ sprintf\ \phi = sprintfk\ (kformat\ \phi)$.

5 Typing formats with type transformers

The previous typing of *kformat* is unfair. If ϕ is a given closed format, the expression *kformat* ϕ has a closed type as well. A limitation of this typing is that it prevents formats to be sequentially composed. For example,

$$(kformat\ (Lit\ foo\ (Str\ Stop)))\ \circ\ (kformat\ (Lit\ bar\ (Int\ Stop))) \tag{15}$$

is ill-typed. In order to recover plain Danvy's functional formats, which do not suffer from such limitations, we use type transformers. In some sense, the latter implement the type inference mechanism of the ML type system. In our example, the type transformer to be considered maps a type α to $string \to int \to \alpha$.

Definition $idt := \lambda\alpha.\ \alpha$.
Definition $datat\ P := \lambda\alpha.\ (X\,P \to \alpha)$.
Fixpoint $type\_transf\_of\_fmt\ \phi : Set \to Set :=$
match ϕ with
| $Stop \Rightarrow idt$
| $Data\ P\ \phi \Rightarrow (datat\ P) \circ (type\_transf\_of\_fmt\ \phi)$
| $Lit\ s\ \phi \Rightarrow type\_transf\_of\_fmt\ \phi$
end.

The new typing of *r_sprintf* is as follows.

Fixpoint $r\_sprintf\ \phi : string \to type\_transf\_of\_fmt\ \phi\ string$:=
match ϕ with
| $Stop \Rightarrow \lambda a.\ a$
| $Data\ P\ \phi \Rightarrow \lambda a\, x.\ r\_sprintf\ \phi\ (a \,\hat{}_P\, x)$
| $Lit\ s\ \phi \Rightarrow \lambda a.\ r\_sprintf\ \phi\ (a\,\hat{}\,s)$
end.

Definition $sprintf\ \phi := r\_sprintf\ \phi$ "".

5.1 Polymorphic functional formats

In this version, the type given to a functional format takes the form *kt tf*, where *tf* is a type transformer.

Definition $kt\ (tf : Set \to Set) :=\ \forall\alpha\ (string \to \alpha) \to string \to\ tf\,\alpha.$

Accordingly, the new typings of *kid*, *kdata* and *lit* are:

Definition $kid\ :\ kt\,idt := \lambda\alpha.\ \lambda k\,a.\ k\,a.$

Definition $kdata\ P\ :\ kt\,(\lambda\alpha.\ X\,P \to \alpha) := \lambda\alpha.\ \lambda k : string \to \alpha.\ \lambda a\,x.\ k\,(a\,\hat{}_P\,x).$

Definition $lit\ x\ :\ kt\ idt := \lambda\alpha.\ \lambda k\,a.\ k\,(a\,\hat{}\,x).$

Observe that, in this version, no additional argument is needed in *lit* and *kdata* (or its instances such as *sint*).

The counterpart of type unification shown in equations (6) to (9) of the introduction is performed in the following version of function composition.

Definition $u\_seq\,(tg, tf : Set \to Set) : kt\,tg \to kt\,tf \to kt\,(tg \circ tf) :=$
$\lambda g f.\ \lambda\alpha.\ \lambda k.\ g\,(tf\,\alpha)\,(f\,\alpha\,k).$

We use the infix notation $\circledcirc$ for *u_seq*.

5.2 Translation

Definition $sprintfk\ (tf : Set \to Set) : kt\,tf \to tf\,string := \lambda f.\,f\,string\,(\lambda s.\,s)$ "".

Fixpoint $kformat\ \phi : kt\ (type\_transf\_of\_fmt\ \phi) :=$
match ϕ with
| $Stop \Rightarrow kid$
| $Data\ P\ \phi \Rightarrow (kdata\ P) \circledcirc (kformat\ \phi)$
| $Lit\ x\ \phi \Rightarrow (lit\ x) \circledcirc (kformat\ \phi)$
end.

As desired, formats can be composed. For example, *kformat example* is convertible with $(kformat\ (Lit\ foo\ (Str\ Stop))) \circledcirc (kformat\ (Lit\ bar\ (Int\ Stop)))$. A format can even be composed with itself, as in

let $kex = kformat\ example$ in $kex \circledcirc kex.$

5.3 Correctness of sprintfk w.r.t. sprintf

The proof is along the same lines as before. In the induction steps, we have to recognize a higher-order pattern involving another kind of function composition, which is defined by $f \circ_2 g\ :=\ \lambda x\,y.\ f\,(g\,x\,y)$.

The two key remarks are:

$$\forall P\quad kdata\,P = \lambda\alpha.\ \lambda k{:}string \to\alpha.\ k \circ_2 (r\_X\ P) \tag{16}$$

and

$$\forall tf{:}\,Set{\to}Set\ \forall f{:}\,kt\,tf\ \forall P\ (kdata\ P) \circledcirc f\ =\ \lambda\alpha.\ \lambda k.\ (f\ \alpha\ k) \circ_2 (r\_X\ P) \tag{17}$$

where = stands for convertibility. We can inline these identities in order to get versions of *r_sprintf* and *kformat* which are convertible with the original ones.

Fixpoint $r\_sprintf1\ \phi : string \rightarrow type\_transf\_of\_fmt\,\phi\ string :=$
match ϕ with
| $Stop \Rightarrow \lambda a.\ a$
| $Data\,P\,\phi \;\Rightarrow\; (r\_sprintf1\,\phi) \circ_2 (r\_X\,P)$
| $Lit\,s\,\phi \;\Rightarrow\; (r\_sprintf1\,\phi) \circ (\lambda a.\ a\hat{\ }s)$
end.

Fixpoint $kformat1\ \phi$: $kt\ (type\_transf\_of\_fmt\ \phi) :=$
match ϕ with
| $Stop \Rightarrow kid$
| $Data\,P\,\phi \;\Rightarrow\; \lambda\alpha.\ \lambda k.\ (kformat1\,\phi\,\alpha\,k) \circ_2 (r\_X\,P)$
| $Lit\,x\,\phi \;\Rightarrow\; \lambda\alpha.\ \lambda k.\ (kformat1\,\phi\,\alpha\,k) \circ (\lambda a.\ a\hat{\ }x)$
end.

Using them, we can prove that:

$$\forall\phi\ \ r\_sprintf\,\phi \;=\; kformat\,\phi\ string\,(\lambda s.\ s) \tag{18}$$

by a straightforward induction over ϕ, and we get the desired theorem in the same way as in section 4.

Theorem *sprintf_sprintfk*: $\forall\phi\ sprintf\,\phi = sprintfk\ (kformat\ \phi)$.

Acknowledment

The work reported here was started during a stay at SRI, thanks to an invitation of N. Shankar and J. Rushby. The first draft of this paper was written in a undisclosable amount of time using the coqdoc tool of J.-C. Filliâtre. I also wish to thank anonymous referees for their constructive comments.

References

1. Olivier Danvy. Functional Unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.
2. The Coq Development Team, LogiCal Project, V8.0. The Coq Proof Assistant Reference Manual. Technical report, INRIA, 2004.