Module combi

1 Functional combinators

Definition $seq\ (A\ B\ C : Type)\ (f : B \rightarrow C)\ (g : A \rightarrow B)\ (x : A) := f\ (g\ x)$.
Implicit Arguments seq $[A\ B\ C]$.

Notation "f 'o' g" $:= (seq\ f\ g)$ (*at level* 5, *right associativity*).

Definition $seq2\ (A\ B\ C\ D : Set)\ (f : D \rightarrow A)\ (g : B \rightarrow C \rightarrow D)$
$(x : B)\ (y : C) := f\ (g\ x\ y)$.
Implicit Arguments seq2 $[A\ B\ C\ D]$.

Notation "f '$o2$' g" $:= (seq2\ f\ g)$ (*at level* 5, *right associativity*).

Lemma rew_apply :
$\forall\ (A\ B : Set)\ (f\ g : A \rightarrow B),\ f = g \rightarrow \forall\ x : A,\ f\ x = g\ x$.
Proof.
intros A B f g E x; case E; reflexivity.
Qed.
Implicit Arguments rew_apply $[A\ B\ f\ g]$.

Module concrete_format

2 Strings

Parameter $string : Set$.

append string
Parameter $aps : string \rightarrow string \rightarrow string$.
Parameter $int : Set$.
Parameter $string\_of\_int : int \rightarrow string$.
Parameter $nullstr : string$.

Notation " '#' " $:= nullstr$.
Notation "u ^^ v" $:= (aps\ u\ v)$ (*at level* 5, *right associativity*).

Definition $r\_int\ a\ i := a$ ^^ $(string\_of\_int\ i)$.

3 Concrete formats

Each type X to be printed is endowed with 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 is handier and yields shorter definitions. Reasoning about functions r_X, rather than on strings, is a simple but important design step of the present developement.

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

Notation "a ^/ P /^ x " $:= (r\_X\ P\ a\ x)$ (*at level* 10, P *at level* 99).

Implicit Arguments mkpr $[X]$.

A printf format specification such as "*foo %s bar %i*" is represented by a concrete value (*Lit foo* (*Str* (*Lit bar* (*Int Stop*)))) and is mapped to the type
$string \rightarrow int \rightarrow A$.

In order to leave the type of printable data types open, specifications such as *Str* and *Int* are not constructors, but special cases of *Data*.

Inductive $format : Type :=$
$|\ Stop : format$
$|\ Data : Printable \rightarrow format \rightarrow format$
$|\ Lit : string \rightarrow format \rightarrow format$.

Examples

Definition $pstr := mkpr\ aps$.
Definition $pint := mkpr\ r\_int$.

Definition $Str := Data\ pstr$.
Definition $Int := Data\ pint$.

Lists can easily be taken into account.

The following tokens represent respectively ',' '[' and ']'.
Parameters comma leftbk rightbk : *string*.

Require Import *List*.

Fixpoint $r\_nelist\ (P : Printable)\ (a : string)\ (l : list\ (X\ P))\ \{struct\ l\}$:
$string :=$
match l with
$|\ nil \Rightarrow a$ ^^ $rightbk$
$|\ x :: l \Rightarrow r\_nelist\ P\ a$ ^^ $(comma$ ^/P/^ $x)\ l$
end.

Definition r_list $(P : Printable)$ $(a : string)$ $(l : list\ (X\ P)) : string :=$
  match l with
  $|\ nil \Rightarrow a$ ^^ $leftbk$ ^^ $rightbk$
  $|\ x :: l \Rightarrow r\_nelist\ P\ a$ ^^ $(leftbk$ ^/P/^ $x)\ l$
  end.

Definition $plist$ $(P : Printable) := mkpr\ (r\_list\ P)$.

Definition Lis $(P : Printable) := Data\ (plist\ P)$.
Definition $Lisi := Lis\ pint$.
Definition $Lisii := Lis\ (plist\ pint)$.

In the sequel we will use the following concrete example.
Parameters foo bar : string.
Definition $example := Lit\ foo\ (Str\ (Lit\ bar\ (Int\ Stop)))$.

Module functional_format

Require Import *combi*.
Require Import *concrete_format*.

4 Typing formats with type transformers

A simpler typing of *kformat* prevents one to compose formats, while there is no such limitation with Damas-Milner typing.Therefore, we will use type transformers. In some sense, the latter show the type inference mechanism of the ML type system at work. The idea is that a functional format maps a type α to $string \rightarrow int \rightarrow \alpha$ (in our example).

4.1 Type transformers

Definition idt $(A : Set) := A$.
Definition $datat$ $(P : Printable)$ $(A : Set) := X\ P \rightarrow A$.
Definition $intt := datat\ pint$.
Definition $strt := datat\ pstr$.

4.2 Type transformer associated to a concrete format

Fixpoint $type\_transf\_of\_fmt$ $(f : format) : Set \rightarrow Set :=$
  match f with
  $|\ Stop \Rightarrow idt$
  $|\ Data\ P\ f \Rightarrow (datat\ P)\ o\ (type\_transf\_of\_fmt\ f)$
  $|\ Lit\ s\ f \Rightarrow type\_transf\_of\_fmt\ f$
  end.

Example (cont'd)

Eval *compute* in $(type\_transf\_of\_fmt\ example)$.
$= \lambda\alpha : Set.\ string \rightarrow int \rightarrow \alpha$

Fixpoint $r\_sprintf$ $(f : format) : string \rightarrow type\_transf\_of\_fmt\ f\ string :=$
  match f return $(string \rightarrow type\_transf\_of\_fmt\ f\ string)$ with
  $|\ Stop \Rightarrow \lambda\ a \Rightarrow a$
  $|\ Data\ P\ f \Rightarrow \lambda\ a\ x \Rightarrow r\_sprintf\ f\ (a$ ^/P/^ $x)$
  $|\ Lit\ s\ f \Rightarrow \lambda\ a \Rightarrow r\_sprintf\ f\ a$ ^^ s
  end.

Definition $sprintf$ $(f : format) := r\_sprintf\ f\ \#$.

4.3 Functional formats

The main argument of *sprintfk* is a function of type $(kt\ tf)$, where tf can be expressed by means of a format.

CPS version of appending to the right.
Definition kt $(tf : Set \rightarrow Set) :=$
  $\forall\ A : Set, (string \rightarrow A) \rightarrow string \rightarrow tf\ A$.

Example (cont'd)

Eval *compute* in $(kt\ (type\_transf\_of\_fmt\ example))$.
$= \forall\ A : Set, (string \rightarrow A) \rightarrow string \rightarrow string \rightarrow int \rightarrow A$

The main argument of *sprintfk* is a functional format f.We only have to know that its type has the form $kt\ tf$.The fact that f may be computed using a concrete formatis not relevant.

Definition $sprintfk$ $(tf : Set \rightarrow Set)$ $(f : kt\ tf) :$
  $tf\ string := f\ string\ (\lambda\ s \Rightarrow s)\ \#$.
Implicit Arguments sprintfk [tf].

Functional formats are constructed using primitives 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 $kid : kt\ idt := \lambda\ A\ k\ a \Rightarrow k\ a$.

Definition $kdata$ $(P : Printable) : kt\ (\lambda\ A \Rightarrow X\ P \rightarrow A) :=$
  $\lambda\ (A : Set)\ (k : string \rightarrow A)\ a\ x \Rightarrow k\ (a$ ^/P/^ $x)$.

Definition lit $(x : string) : kt\ idt := \lambda\ A\ k\ a \Rightarrow k\ (a$ ^^ $x)$.

4.4 Examples

Definition $str := kdata\ pstr$.
Definition $sint := kdata\ pint$.

Definition $lis\ (P : Printable) := kdata\ (plist\ P)$.
Definition $lisi := lis\ pint$.
Definition $lisii := lis\ (plist\ pint)$.

4.5 Translation

Composition of polymorphic functions: 'o' cannot be used, though the underlying untyped term is the same. The type of the result is explicitly computedand put in the form $(kt\ \tau)$, where τ happens to beobtained by sequential composition of type transformers.In this way, the job of Damas-Milner type inference is done explicitly.

Definition $seqp\ (tg\ tf : Set \to Set)\ (kg : kt\ tg)\ (kf : kt\ tf)$:
$kt\ (tg\ o\ tf) := \lambda\ A\ k \Rightarrow kg\ (tf\ A)\ (kf\ A\ k)$.
Implicit Arguments seqp [*tg tf*].
Notation "*f 'oo' g*" := (*seqp f g*) (*at level* 5, *right associativity*).

Fixpoint $kformat\ (f : format) : kt\ (type\_transf\_of\_fmt\ f) :=$
match f return $(kt\ (type\_transf\_of\_fmt\ f))$ with
| $Stop \Rightarrow kid$
| $Data\ P\ f \Rightarrow (kdata\ P)\ oo\ (kformat\ f)$
| $Lit\ s\ f \Rightarrow (lit\ s)\ oo\ (kformat\ f)$
end.

Example (cont'd)

Remark $check\_value : kformat\ example = (lit\ foo)\ oo\ str\ oo\ (lit\ bar)\ oo\ sint$.
Proof. *reflexivity*. Qed.

Formats can be composed.
Remark $check\_compo$:
$kformat\ example =$
$(kformat\ (Lit\ foo\ (Str\ Stop)))\ oo\ (kformat\ (Lit\ bar\ (Int\ Stop)))$.
Proof. *reflexivity*. Qed.

A format can even be composed with itself.
Check (*let kex* := *kformat example* in *kex oo kex*).

4.6 Correctness of sprintfk w.r.t. sprintf

A brutal attempt to prove that $(sprintf\ f) = (sprintfk\ (kformat\ f))$ holds for all f fails, because the accumulator changes at each recursive call (an induction on f would lead us to

to prove something on $\#\ \hat{}\hat{}\ s$ 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

$$\forall\ a\ (r\_sprintf\ f\ a) = (kformat\ f\ (\lambda\ s \Rightarrow s)\ a) \tag{1}$$

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

$$\lambda\ x \Rightarrow r\_sprintf\ f\ (a\ \hat{}/P/\hat{}\ x)\ =\ \lambda\ x \Rightarrow kformat\ f\ string\ (\lambda\ s \Rightarrow s)\ (a\ \hat{}/P/\hat{}\ x)$$

from the induction hypothesis (1)? 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
$extensionality$:
$\forall\ (A\ B : Set)\ (f\ g : A \to B)$,
$(\forall\ x : A, f\ x = g\ x) \to (\lambda\ x \Rightarrow f\ x) = (\lambda\ x \Rightarrow g\ x)$.

But this workaround is not satisfactory. A first way out is to prove a weakened extensional version of the theorem, informally telling us that applying $(r\_sprintf\ f\ a)$ to its arguments yields the same string as $(kformat\ f\ string\ (\lambda\ s \Rightarrow s)\ a)$. This is a somewhat tedious exercise in CIC. We just need formal definitions of:

- an arguments vector paramerized by a format;
- the application of dependent functions like $(r\_sprintf\ f\ a)$ to such a vector.

4.6.1 Intensionalization

However, we can prove the first formulation of the theorem,provided we work with a *still more intensional* statement:

$$\lambda\ a \Rightarrow r\_sprintf\ f\ a\ =\ \lambda\ a \Rightarrow kformat\ f\ (\lambda\ s \Rightarrow s)\ a \tag{2}$$

We call (2) the *intensionalized* version of (1).More generaly, the intensionalized version of $\forall x,\ P = Q$ is $\lambda x\, P = \lambda x\, Q$.

The proof is by induction on the format f. In the second case, unfolding *kdata* and *oo* in *kformat* yields something similar to what we have in *r_sprintf* (A and k will be respectively equal to *string* and $\lambda\ x \Rightarrow x$).

Remark $key\_data\_kformat$:
$\forall\ A\ k\ (tf : Set \to Set)\ (kf : kt\ tf)\ (P : Printable)$,
$((kdata\ P)\ oo\ kf)\ A\ k = \lambda\ a\ x \Rightarrow kf\ A\ k\ (a\ \hat{}/P/\hat{}\ x)$.
Proof. *reflexivity*. Qed.

This can be expressed with the induction hypothesis $\lambda\ a \Rightarrow g\ a$ using the combinator *o2*.

Remark *key_data* :
$\forall\ P\ (A{:}Set)\ (g{:}string{\rightarrow}A),$
$(\lambda\ a\ x \Rightarrow g\ (a\ \hat{}/P/\hat{}\ x)) = (\lambda\ a \Rightarrow g\ a)\ o2\ (r\_X\ P).$
Proof. *reflexivity.* Qed.

And similarly in the third induction case, with *o* instead of *o2*.
Remark *key_lit_kformat* :
$\forall\ A\ k\ (tf : Set \rightarrow Set)\ (kf : kt\ tf)\ (x : string),$
$((lit\ x)\ oo\ kf)\ A\ k = \lambda\ a{\Rightarrow}\ kf\ A\ k\ (a\ \hat{}\hat{}\ x).$
Proof. *reflexivity.* Qed.

Remark *key_lit* :
$\forall\ s\ A\ (g{:}string{\rightarrow}A),$
$(\lambda\ a \Rightarrow g\ (a\ \hat{}\hat{}\ s)) = (\lambda\ a \Rightarrow g\ a)\ o\ (\lambda\ a{\Rightarrow}\ a\ \hat{}\hat{}\ s).$
Proof. *reflexivity.* Qed.

Using these remarks we get a straightforward proof. We can also inline them in order to get intensionally equal definitions of r_sprintf and kformat.

```
Fixpoint r_sprintf1 (f : format) : string → type_transf_of_fmt f string :=
  match f return (string → type_transf_of_fmt f string) with
  | Stop ⇒ λ a⇒ a
  | Data P f ⇒ (λ a ⇒ (r_sprintf1 f a)) o2 (r_X P)
  | Lit s f ⇒ (λ a ⇒ (r_sprintf1 f a)) o (λ a⇒ a ^^ s)
  end.
```

For *kformat* we actually inline intensionalized versions of *key_data_kformat* and of *key_lit_kformat*.

```
Fixpoint kformat1 (f : format) : kt (type_transf_of_fmt f) :=
  match f return (kt (type_transf_of_fmt f)) with
  | Stop ⇒ kid
  | Data P f ⇒ λ A k⇒ (λ a ⇒ (kformat1 f A k a)) o2 (r_X P)
  | Lit s f ⇒ λ A k⇒ (λ a ⇒ (kformat1 f A k a)) o (λ a⇒ a ^^ s)
  end.
```

```
Theorem sprintf_kformat :
 ∀ f : format,
   (λ a ⇒ r_sprintf f a) = (λ a ⇒ kformat f string (λ s⇒ s) a).
Proof.
  change (∀ f : format,
   (λ a ⇒ r_sprintf1 f a) = (λ a ⇒ kformat1 f string (λ s⇒ s) a)).
  simple induction f; clear f; simpl.
    unfold kid. reflexivity.
    intros P f hrec. case hrec. reflexivity.
    intros x f hrec. case hrec. reflexivity.
```


Qed.

4.6.2 Simplification

By chance, both members of the desired equality happen to have the form $\lambda a.\, ga$, hence we can try a still more intensional version, i.e. an η-reduced version of (2):

$$r\_sprintf\ f\ =\ kformat\ f\ (\lambda\ s{\Rightarrow}\ s). \tag{3}$$

The proof is straightforward, we even don't need to recognize combinatorial expressions.

```
Theorem sprintf_kformat_eta :
 ∀ f : format, r_sprintf f = kformat f string (λ s⇒ s).
Proof.
  simple induction f; clear f; simpl.
    unfold kid. reflexivity.
    intros P f hrec. unfold seqp, kdata. case hrec. reflexivity.
    intros s f hrec. unfold seqp, lit. case hrec. reflexivity.
Qed.
```

4.6.3 Wanted corollary

Remark that the concrete format f is *not* an argument of *sprintfk*.

```
Theorem sprintf_sprintfk :
 ∀ f : format, sprintf f = sprintfk (kformat f).
Proof.
  intro f. unfold sprintf, sprintfk.
  exact (rew_apply (sprintf_kformat_eta f) #).
Qed.
```