Composition & Deforestation *or* how to provide Genericity ?

Loïc Correnson - Etienne Duris
Didier Parigot - Gilles Roussel

INRIA Rocquencourt

`http://www-rocq.inria.fr/oscar/`

$\boxed{\text{Genericity}}$

How to cook something with a pierced frying pan ?

- Repair the frying pan,
- Take the “something” to cook, and put it in the pan,
- Find in a book the cooking time,
- Put the pan on the fire and wait.

$\boxed{\text{Reusability}}$

How to cook an egg with a frying pan ?

– Pierce it, and refer to the previous slide.

$\boxed{\text{Instanciation}}$

How to cook an egg with a frying pan ?

- Pierce it,

- Repair the frying pan,

- Take the **egg** to cook, and put it in the pan,

- Find in a book the cooking time,

- Put the pan on the fire and wait.

$\boxed{\text{Deforestation}}$

How to cook an egg with a frying pan ?

- Take the **egg** to cook, and put it in the pan,

- Put the pan on the fire and wait two minutes.

A Position for Genericity

Our position about genericity is :

- Reuse known algorithms by translations and standard composition.

- Implement abstract operators with base-level constructions, rather than designing new super-languages.

- Perform the actual instanciation of known algorithms and abstract operators by general program transformations.

$\boxed{\text{Two examples}}$

Reusing : unification

Abstracting : polytypic

$\boxed{\textbf{Instanciation = Deforestation !}}$

First example : unification

A very convenient date type for unification is :

$$\texttt{type}\ term\ \ c\ \ v = cons\ \ c\ \ (list\ \ term)\ |\ var\ \ v$$

Then the unification of two terms is a function of the following type :

$$unif\ : term\ \ c\ \ v \to term\ \ c\ \ v \to (v \to term\ \ c\ \ v)$$

and raises an exception if unification is impossible.

First example : unification

With two coupling functions, $unif$ could be instanciated on a type τ :

$$abstract : \tau \rightarrow term\ \tau_c\ \tau_v$$
$$concrete : term\ \tau_c\ \tau_v \rightarrow \tau$$

$unif\_tau\ \ a\ \ b =$
  `let` $s = unif\ \ (abstract\ \ a)\ (abstract\ \ b)$ `in`
  $\lambda x.(concrete\ \ (s\ \ x))$

Polytypic Programming

– A genericity based on functors.

– Functors describe types and functions recursion's,

– Provides high-level operators constructed with low-level mathematical ones.

Functors

They define recursive types and could be applied to functions and types.

For instance :

$$\texttt{type}\ list\ \ a = nil \mid cons\ \ a\ (list\ \ a)$$
$$\mathcal{F}_{list} = 1 + \texttt{Par} * \texttt{Rec}$$
$$\mathcal{F}_{list}\ f\ \ g = \texttt{id}\ + (f * g)$$
$$\mathcal{F}_{list}\ \tau_1\ \ \tau_2 = inl\ () \mid inr\ (\tau_1, \tau_2)$$

$\boxed{\text{Isomorphisms}}$

Translations from polymorphic types :

$$\tau\ \alpha \rightarrow \begin{cases} \mathcal{F}_\tau \\ \mathtt{inn}_\tau\ : \tau\ \alpha \leftarrow (\mathcal{F}_\tau\ \alpha\ (\tau\ \alpha)) \\ \mathtt{out}_\tau\ : \tau\ \alpha \rightarrow (\mathcal{F}_\tau\ \alpha\ (\tau\ \alpha)) \end{cases}$$

$\boxed{\text{Mathematical operators (2)}}$

Base level : $(\mathcal{F}\ f\ \ g)$

High level : $pmap\ f$, $cata\ f$

$$
\begin{aligned}
pmap\ f\ \ (x\ :\tau\ \ \alpha) &= \mathtt{inn}_{\tau}\ (\mathcal{F}_{\tau}\ f\ \ (pmap\ f)\ (\mathtt{out}_{\tau}\ x)) \\
cata\ f\ \ (x\ :\tau\ \ \alpha) &= f\ \ (\mathcal{F}_{\tau}\ \mathtt{id}\ \ (cata\ f)\ (\mathtt{out}_{\tau}\ x))
\end{aligned}
$$

$\boxed{\text{How to implement Polytypic Programming ?}}$

$$\begin{array}{ccc} \tau\ \alpha & \leftrightarrow & F\ \tau\ (\tau\ \alpha) \\ \mathbf{x} & \longrightarrow & out(\mathbf{x}) \\ inn(\mathbf{y}) & \longleftarrow & \mathbf{y} \end{array}$$

Thus : $F = (inn, out)$.

Need special constructors to represent the type $(F\ \tau_1\ \tau_2)$.

$\boxed{\text{An alternative to the Super-Language}}$

Special constructors as a concrete type:

$$
\begin{array}{lcl}
\texttt{type}\ \mathit{Functor}\ a\ b & = & (b \to a) * (a \to b) \\
\texttt{type}\ \mathit{Regular}\ a\ b\ c & = & \texttt{Sum}\ \mathit{int}\ \mathit{Regular} \\
 & | & \texttt{Prod}\ \mathit{Regular}\ \mathit{Regular} \\
 & | & \texttt{Par}\ a \\
 & | & \texttt{Rec}\ b \\
 & | & \texttt{Const}\ c
\end{array}
$$

$\boxed{\text{Functor definition :}}$

$$
\begin{array}{l}
\mathit{functor\_prod}\ (i,o)\ (i',o') = \\
\quad \texttt{let}\ i'' = \lambda x.\texttt{match}\ x\ \texttt{with}\ \texttt{Prod}\ a\ \ b \rightarrow (i(a), i'(b))\ \texttt{in} \\
\quad \texttt{let}\ o'' = \lambda x.\texttt{match}\ x\ \texttt{with}\ (a,b) \rightarrow \texttt{Prod}(o(a), o'(b))\ \texttt{in} \\
\quad (i'', o'') \\
\mathit{functor\_sum} \ldots
\end{array}
$$

$$
\begin{array}{l}
\mathit{fmap}\ f\ \ g = \lambda x.\texttt{match}\ x\ \texttt{with} \\
\quad \texttt{Sum}\ c\ \ y \longrightarrow \texttt{Sum}\ c\ (\mathit{fmap}\ \ g\ \ y) \\
\quad \texttt{Prod}\ y\ \ z \longrightarrow \texttt{Prod}\ (\mathit{fmap}\ \ g\ \ y)\ (\mathit{fmap}\ f\ \ g\ \ z) \\
\quad \texttt{Par}\ a \longrightarrow \texttt{Par}\ (f\ \ a) \\
\quad \texttt{Rec}\ b \longrightarrow \texttt{Rec}\ (g\ \ b) \\
\quad \texttt{Const}\ c \longrightarrow \texttt{Const}\ c
\end{array}
$$

$\boxed{\text{The library}}$

(How to cook something with a pierced span)

$$
\begin{aligned}
pmap\ (i,o)\ f &= i.(fmap\ f\ \ (pmap\ (i,o)\ f)).o \\
cata\ (i,o)\ f\ \ x &= f.(fmap\ id\ (cata\ (i,o)\ f)).o \\
ana\ (i,o)\ f\ \ x &= i\ .(fmap\ id\ (ana\ (i,o)\ f)).f
\end{aligned}
$$

etc.

$\boxed{\text{The generated coupling}}$

(How to cook an **egg**. . .)

$$\mathit{out\_list}\ (\mathit{cons}\ \ a\ \ b) = (\texttt{Sum}\ \mathit{1}\ (\texttt{Prod}\ (\texttt{Par}\ a)\ (\texttt{Rec}\ b)))$$
$$\mathit{out\_list}\ (\mathit{nil}) = (\texttt{Sum}\ \mathit{2}\ (\texttt{Const}())$$

$$\mathit{inn\_list}\ (\texttt{Sum}\ \mathit{1}\ (\texttt{Prod}\ (\texttt{Par}\ a)\ (\texttt{Rec}\ b))) = (\mathit{cons}\ \ a\ \ b)$$
$$\mathit{inn\_list}\ (\texttt{Sum}\ \mathit{2}\ (\texttt{Const}()) = (\mathit{nil})$$
$$\mathit{inn\_list}\ \_ = \texttt{raise Error}$$

$$\mathit{F\_list} = (\mathit{inn\_list}\ \ ,\ \ \mathit{out\_list})$$

$\boxed{\text{The instanciation}}$

(. . . with a frying pan ?)

(pierce it !)

$$pmap\_list = pmap \;\; F\_list$$

Few steps of the deforestation process

partial evaluation :

$$pmap\_list\ f = inn\_list.(fmap\ f\ (pmap\_list\ f)).out\_list$$

deforestation :

$$\begin{array}{c}(cons\ a\ b)\\ (Sum\ 2\ (Prod\ (Par\ a)\ (Rec\ b)))\\ (Sum\ 2\ (Prod\ (Par\ (f\ a))\ (Rec\ (pmap\_list\ f\ b))))\\ (cons\ (f\ a)\ (pmap\_list\ f\ b))\end{array}$$

The result after deforestation

$$
\begin{aligned}
&pmap\_list\ f\ x = g\ f\ x\\
&g\ f\ x = \texttt{match}\ x\ \texttt{with}\\
&\quad cons\ a\ b \rightarrow cons\ (f\ a)\ (g\ f\ b)\\
&\quad nil \rightarrow nil
\end{aligned}
$$

$\boxed{\text{Deforestation : a known problem}}$

Philip Wadler, 1988

HYLO system, 1997

Symbolic Composition, 1998

$\boxed{\text{Conclusion}}$

– Reuse is specified by composition with coupling functions.

– Deforestation could perform the specialization.

– High-level operations (PolyP) ought to be simulated at base-level (Haskell), and deforested w.r.t the instanciation.