Labeled functions in OCaml — categorically

Mikołaj Konarski
mikon@mimuw.edu.pl

April 1, 2003

Several years ago, while designing a categorical programming language, I've spent weeks deciding which categorical construction would best model free variables and the mechanism of abstraction. With the formalism I present here, these analyses are much easier to do and perhaps even presentable during one talk.

A year ago "labels" has been added to OCaml, which seem to be almost the same mechanism as the categorical construction I've adopted for my language. This coincidence is great, since I've got a nice syntax for free.

But is this a coincidence? How exact are the similarities? Is the labeled exponent of my language an ad hoc construction, as opposed to the standard exponential object? Could my categorical considerations result in a different abstraction mechanism with acceptable programming properties? I will deploy the basics of the categorical machinery of my Ph.D. thesis to answer these questions.

These are toy questions. But more or less the same machinery may serve to define simple semantics of sophisticated programming modules, to investigate another adjunctions, for example the coexponent (modeling the exception or even continuation mechanisms), and to analyze rewriting systems for (co)inductive datatypes, as well as the limits of their applicability.

A functional story

In OCaml there are "labels": there are labeled function types

```
sig
  val add : x:int -> y:int -> int
  val fold : f:(acc:'a -> elem:'b -> 'a) ->
             init:'a ->
             l:'b list -> 'a
  val sublist : l:'a list -> pos:int -> len:int -> `a list
end
```

and labeled functions themselves

```
  let add = fun ~x ~y -> x + y
  let fold = fun ~f ~init ~l -> ...
  let sublist = fun ~l ~pos ~len -> ...
```

which can be applied

```
  let a = add ~x:5 ~y:10
  let b = add ~y:10 ~x:5
  let c = let x = 5 in
          let z = 10 in
          add ~x ~y:z
  let prefix = sublist ~pos:1
  let l56 = prefix ~len:2 ~l:[5, 6, 7]
  let nth = sublist ~len:1
```

There is much more syntactic sugar and other additional features.

The thing to note is that the labeled functions are not invariant under α-conversion of their labels. So, for example, the following two declarations are not type-correct

```
let v = let add10 = fun ~x -> x + 10 in
        add10 ~a:5
let w = ((fun ~a ~b -> a + b) : x:int -> y:int -> int)
```

That there is no α-conversion seems to indicate this construction is ad hoc. But there is another programming language construct that has no α-conversion, namely records.

```
let r = ({a = 1; b = "baba"} : {a : int; b : string})
```

The labels are not bound in records nor in labeled functions. There are still *free variables* in the language, but no longer there are *captured variables*.

If a language has records then the labeled functions may be expressed using ordinary functions and records.

```
let add = fun (arg : {x : int; y : int}) -> arg.x + arg.y
let fifteen = add {x = 5; y = 10}
```

This still suggests that labeled functions are ad hoc, this time that they are an ad hoc abbreviation. Nevertheless, for safety and efficiency reasons and to clarify the semantics of extensions they are best considered as a separate facet of the language. So for example the following declarations are illegal in OCaml.

```
let wrong15 = (fun arg -> arg.x + arg.y) ~x:5 ~y:10
let wrong10 = (fun ~x ~y -> x + y) {x = 5; y = 5}
```

In fact there is an independent, fundamental meaning to the labeled functions datatype, one that supersedes the semantics derived from the abbreviation.

A categorical story

Let us look closer at some adjunctions in **Cat**. To simplify matters let us concentrate on functors going between category **Set** and its products.

Let $\Delta : \mathbf{Set} \to \mathbf{Set} \times \mathbf{Set}$ be a functor given by $A \mapsto \texttt{<}A, A\texttt{>}$. A right adjoint to Δ is a functor that maps two sets to their binary product (for example to their cartesian product). Let us call one such functor $\_ * \_ : \mathbf{Set} \times \mathbf{Set} \to \mathbf{Set}$.

The whole right adjunction to Δ may be thought of as providing the binary product datatype. On the other hand the left adjunction to Δ provides the disjoint sum datatype. Let B be a set and let $\mathbf{X}_B : \mathbf{Set} \to \mathbf{Set}$ be a functor given by $A \mapsto A * B$. The right adjoint to $\mathbf{X}_B$ maps set C to an exponential object of B and C (for example the function space $[B \to C]$). Thus the right adjunction to $\mathbf{X}_B$ provides the functional datatype.

Whoa! How wonderful the most important datatypes get generated by adjunctions using only the products in **Cat**. Now, what would that be, if we start not from binary products in **Cat** but from finite ones and even better from labeled products, analogous to record types of programming languages? Let us start again.

Let $\Delta : \mathbf{Set} \to \texttt{<}i_1 \texttt{ - } \mathbf{Set}\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ - } \mathbf{Set}\texttt{>}$ be a functor given by $A \mapsto \texttt{<}i_1 \texttt{ : } A\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } A\texttt{>}$. It turns out that the right adjunction to Δ provides the record datatype. Let the functor providing the type be denoted by $\texttt{\{}i_1 \texttt{ : } \_\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } \_\texttt{\}}$. On the other hand the left adjunction to Δ provides the variant datatype, similar to the one of OCaml or SML.

Let us try to devise something similar to the functional datatype using the above functors. Let $B, B_1, \ldots, B_n$ be sets. Should we be obtaining right adjunction to the functor given by $A \mapsto \texttt{\{}j \texttt{ : } A\texttt{;}\ i_1 \texttt{ : } B\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } B\texttt{\}}$, or $A \mapsto \texttt{\{}j \texttt{ : } A\texttt{;}\ i_1 \texttt{ : } B_1\texttt{;}\ \ldots\texttt{;}\ i_n \texttt{ : } B_n\texttt{\}}$? Or perhaps we should use $A \mapsto \texttt{\{}j_1 \texttt{ : } A\texttt{;}\ \ldots\texttt{;}\ j_n \texttt{ : } A\texttt{;}\ i \texttt{ : } B\texttt{\}}$? Or maybe $\texttt{<}j_1 \texttt{ : } A_1\texttt{;}\ \ldots\texttt{;}\ j_n \texttt{ : } A_n\texttt{>} \mapsto \texttt{\{}j_1 \texttt{ : } A_1\texttt{;}\ \ldots\texttt{;}\ j_n \texttt{ : } A_n\texttt{;}\ i \texttt{ : } B\texttt{\}}$?

We will evaluate the resulting adjunctions from the categorical perspective and the resulting datatypes from a programming perspective. Not surprisingly the datatype that will emerge as the most sensible of all will correspond to the type of OCaml labeled functions. In fact, in the presentation we will choose, the actual construction is slightly more general than the OCaml one, with no apparent programming meaning to that generality.

The link between the nature and the theory

We give a presentation of simply typed lambda-calculus with type variables and finite product types.

First we present the grammar:

```
X ::= a | b | x | y | i | j | ...
C ::= *
F ::= X | {X : F; ...} | F -> F
T ::= X | T . X | {X = T; ...} | T T | fun X -> T
```

Usually, the typing sequents have the following form.

$$\mathtt{X : F}, \ldots \vdash \mathtt{T : F}$$

For example:

$$x \mathbin{\mathtt{:}} \mathtt{\{}i \mathbin{\mathtt{:}} a\mathtt{\}},\ y \mathbin{\mathtt{:}} b \vdash x \mathbin{\mathtt{.}} i \mathbin{\mathtt{:}} a$$

We will also consider sequents for assigning kinds to types.

$$\mathtt{X - C}, \ldots \vdash \mathtt{F - C}$$

For example:

$$a \mathbin{\mathtt{-}} \mathtt{*},\ b \mathbin{\mathtt{-}} \mathtt{*} \vdash \mathtt{\{}j \mathbin{\mathtt{:}} b\mathtt{\}} \mathbin{\mathtt{-}} \mathtt{*}$$

Now we will extend the grammar:

```
C ::= ... | <X - C; ...>
F ::= ... | F . X | <X : F; ...>
T ::= ... | T * <X> | <X = T; ...>
```

With this extension we will prefer the following form of the sequents

$$\mathtt{F} \vdash \mathtt{T : F}$$

and

$$\mathtt{C} \vdash \mathtt{F - C}$$

The example sequents in the original form above now may be written as follows

$$\mathtt{<}x \mathbin{\mathtt{:}} \mathtt{\{}i \mathbin{\mathtt{:}} a\mathtt{\}};\ y \mathbin{\mathtt{:}} b\mathtt{>} \vdash x \mathbin{\mathtt{.}} i \mathbin{\mathtt{:}} a$$

and

$$\mathtt{<}a \mathbin{\mathtt{-}} \mathtt{*};\ b \mathbin{\mathtt{-}} \mathtt{*>} \vdash \mathtt{\{}j \mathbin{\mathtt{:}} b\mathtt{\}} \mathbin{\mathtt{-}} \mathtt{*}$$

We present some of the typing rules for our variant of simply typed λ-calculus. The rules are written using the new form of sequents and writing i, j for elements of $\texttt{X}$, c, d, e for elements of $\texttt{C}$, f, g, h for elements of $\texttt{F}$ and t, u for elements of $\texttt{T}$. First all of the rules for assigning kinds to types.

$$\frac{}{\texttt{<}i\ \texttt{-}\ c\texttt{;}\ \ldots\texttt{>} \vdash i\ \texttt{-}\ c} \tag{1}$$

$$\frac{c \vdash f_1\ \texttt{-}\ \texttt{*} \quad \cdots \quad c \vdash f_n\ \texttt{-}\ \texttt{*}}{c \vdash \texttt{\{}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ f_n\texttt{\}}\ \texttt{-}\ \texttt{*}} \tag{2}$$

$$\frac{c \vdash f\ \texttt{-}\ \texttt{*} \qquad c \vdash g\ \texttt{-}\ \texttt{*}}{c \vdash (f\ \texttt{->}\ g)\ \texttt{-}\ \texttt{*}} \tag{3}$$

$$\frac{c \vdash f\ \texttt{-}\ \texttt{<}i\ \texttt{-}\ d\texttt{;}\ \ldots\texttt{>}}{c \vdash f\ \texttt{.}\ i\ \texttt{-}\ d} \tag{4}$$

$$\frac{c \vdash f_1\ \texttt{-}\ e_1 \quad \cdots \quad c \vdash f_n\ \texttt{-}\ e_n}{c \vdash \texttt{<}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ f_n\texttt{>}\ \texttt{-}\ \texttt{<}i_1\ \texttt{-}\ e_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{-}\ e_n\texttt{>}} \tag{5}$$

Now some of the rules for assigning types to values.

$$\frac{}{\texttt{\{}i\ \texttt{:}\ f\texttt{;}\ \ldots\texttt{\}} \vdash i\ \texttt{:}\ f} \tag{6}$$

$$\frac{f \vdash t\ \texttt{:}\ \texttt{\{}i\ \texttt{:}\ h\texttt{;}\ \ldots\texttt{\}}}{f \vdash t\ \texttt{.}\ i\ \texttt{:}\ h} \tag{7}$$

$$\frac{f \vdash t_1\ \texttt{:}\ h_1 \quad \cdots \quad f \vdash t_n\ \texttt{:}\ h_n}{f \vdash \texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}}\ \texttt{:}\ \texttt{\{}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ h_n\texttt{\}}} \tag{8}$$

$$\frac{f \vdash t\ \texttt{:}\ h}{f\ \texttt{.}\ i \vdash t * \texttt{<}i\texttt{>}\ \texttt{:}\ h\ \texttt{.}\ i} \tag{9}$$

$$\frac{f_1 \vdash t_1\ \texttt{:}\ h_1 \quad \cdots \quad f_n \vdash t_n\ \texttt{:}\ h_n}{\texttt{<}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{>} \vdash \texttt{<}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{>}\ \texttt{:}\ \texttt{<}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{>}} \tag{10}$$

Then we change the form of sequents to be the following.

$$T \triangleright F \to F$$

and

$$F \triangleright C \to C$$

so that the original sequents

$$x \mathtt{\ :\ } \mathtt{\{}i \mathtt{\ :\ } a\mathtt{\}},\ y \mathtt{\ :\ } b \vdash x \mathtt{\ .\ } i \mathtt{\ :\ } a$$

and

$$a \mathtt{\ -\ } *,\ b \mathtt{\ -\ } * \vdash \mathtt{\{}j \mathtt{\ :\ } b\mathtt{\}} \mathtt{\ -\ } *$$

from above translate to

$$x \mathtt{\ .\ } i \triangleright \mathtt{<}x \mathtt{\ :\ } \mathtt{\{}i \mathtt{\ :\ } a\mathtt{\}}\mathtt{;}\ y \mathtt{\ :\ } b\mathtt{>} \to a$$

and

$$\mathtt{\{}j \mathtt{\ :\ } b\mathtt{\}} \triangleright \mathtt{<}a \mathtt{\ -\ } *\mathtt{;}\ b \mathtt{\ -\ } *\mathtt{>} \to *$$

Moreover we add one more projection, three identities and three compositions to the calculus. In this way both value substitution and type instantiation becomes internalizable, as will be seen later.

```
F ::= ... | F_ID(C) | F . F
T ::= ... | <X> | T_ID(C) | T * T | T_id(F) | T . T
```

Using the new composition of values we can write, for example:

```
let v15 = {x = 5; y = 10} . (x + y)
let fxy = {z = v15 - v15} .
  fun ~a -> if z = 0 then a . {xy = x + y; b = 1} else {xy = z; b = 0}
let r = (fxy ~a:{x = 5; y = 10}) . xy
```

Now there are three ambiguities in this grammar, but these are harmless as the following typing rules for the calculus and its equational theory show. The operations of projecting are expressible using the composition and the projections in the following way:

$$\begin{aligned} f \mathtt{\ .\ } i &= f \mathtt{\ .\ } i \\ t \mathtt{\ .\ } i &= t \mathtt{\ .\ } i \\ t * \mathtt{<}i\mathtt{>} &= t * \mathtt{<}i\mathtt{>} \end{aligned}$$

where the terms on the left hand sides are the operations of projecting while the terms on the right hand sides are the compositions.

The calculus will here be presented in a more categorical notation. The whole grammar looks as follows.

```
X ::= a | b | x | y | i | j | ...
C ::= * | <X - C; ...>
F ::= X | {X : F; ...} | F -> F
    | F . X | <X : F; ...>
    | F_ID(C) | F . F
T ::= X | T . X | {X = T; ...} | T T | fun X -> T
    | T * <X> | <X = T; ...> |
    | <X> | T_ID(C) | T * T | T_id(F) | T . T
```

We write i, j for elements of `X`, c, d, e for elements of `C`, f, g, h for elements of `F` and t, u for elements of `T`.

First the rules for assigning kinds to types.

$$\frac{}{i \rhd \texttt{<}i\ \texttt{-}\ c\texttt{;}\ \ldots\texttt{>} \to c} \tag{11}$$

$$\frac{f_1 \rhd c \to \texttt{*} \qquad \cdots \qquad f_n \rhd c \to \texttt{*}}{\texttt{\{}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ f_n\texttt{\}} \rhd c \to \texttt{*}} \tag{12}$$

$$\frac{f \rhd c \to \texttt{*} \qquad g \rhd c \to \texttt{*}}{(f\ \texttt{->}\ g) \rhd c \to \texttt{*}} \tag{13}$$

$$\frac{f \rhd c \to \texttt{<}i\ \texttt{-}\ d\texttt{;}\ \ldots\texttt{>}}{f\ \texttt{.}\ i \rhd c \to d} \tag{14}$$

$$\frac{f_1 \rhd c \to e_1 \qquad \cdots \qquad f_n \rhd c \to e_n}{\texttt{<}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ f_n\texttt{>} \rhd c \to \texttt{<}i_1\ \texttt{-}\ e_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{-}\ e_n\texttt{>}} \tag{15}$$

$$\frac{}{\texttt{F\_ID}(c) \rhd c \to c} \tag{16}$$

$$\frac{f_1 \rhd c \to d \qquad f_2 \rhd d \to e}{f_1\ \texttt{.}\ f_2 \rhd c \to e} \tag{17}$$

Now the rules for assigning types to values.

$$\frac{}{i \rhd \texttt{\{}i\ \texttt{:}\ f\texttt{;}\ \ldots\texttt{\}} \to f} \tag{18}$$

$$\frac{t \rhd f \to \texttt{\{}i\ \texttt{:}\ h\texttt{;}\ \ldots\texttt{\}}}{t\ \texttt{.}\ i \rhd f \to h} \tag{19}$$

$$\frac{t_1 \rhd f \to h_1 \quad \cdots \quad t_n \rhd f \to h_n}{\texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}} \rhd f \to \texttt{\{}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ h_n\texttt{\}}} \tag{20}$$

$$\frac{t \rhd f \to (g\ \texttt{->}\ h) \qquad u \rhd f \to g}{t\ u \rhd f \to h} \tag{21}$$

$$\frac{t \rhd \texttt{\{}j\ \texttt{:}\ f\texttt{;}\ i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ g_n\texttt{\}} \to h}{\texttt{fun}\ j\ \texttt{->}\ t \rhd \texttt{\{}i_1\ \texttt{:}\ g_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ g_n\texttt{\}} \to (f\ \texttt{->}\ h)} \tag{22}$$

$$\frac{t \rhd f \to h}{t * \texttt{<}i\texttt{>} \rhd f\ \texttt{.}\ i \to h\ \texttt{.}\ i} \tag{23}$$

$$\frac{t_1 \rhd f_1 \to h_1 \quad \cdots \quad t_n \rhd f_n \to h_n}{\texttt{<}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{>} \rhd \texttt{<}i_1\ \texttt{:}\ f_1\texttt{;}\ \ldots\texttt{>} \to \texttt{<}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{>}} \tag{24}$$

$$\frac{}{\texttt{<}i\texttt{>} \rhd i \to i} \tag{25}$$

$$\frac{}{\texttt{T\_ID}(c) \rhd \texttt{F\_ID}(c) \to \texttt{F\_ID}(c)} \tag{26}$$

$$\frac{t_1 \rhd f_1 \to h_1 \qquad t_2 \rhd f_2 \to h_2}{t_1 * t_2 \rhd f_1\ \texttt{.}\ f_2 \to h_1\ \texttt{.}\ h_2} \tag{27}$$

$$\frac{}{\texttt{T\_id}(f) \rhd f \to f} \tag{28}$$

$$\frac{t \rhd f \to g \qquad u \rhd g \to h}{t\ \texttt{.}\ u \rhd f \to h} \tag{29}$$

As promised we will show how to express value substitution and type instantiation inside this framework.

For values t and u with the following source and target types

$$\begin{aligned} t &: \{j_1 : g_1;\ \ldots;\ j_n : g_n\} \to f \\ u &: \{i : f;\ j_1 : g_1;\ \ldots;\ j_n : g_n\} \to h \end{aligned}$$

we will write the value substitution as

$$u[t/i] : \{j_1 : g_1;\ \ldots;\ j_n : g_n\} \to h$$

to mean the composition

$$\{i = t;\ j_1 = j_1;\ \ldots;\ j_n = j_n\}\ .\ u$$

For types f and g with the following source and target kinds

$$\begin{aligned} f &: \langle j_1 - c_1;\ \ldots;\ j_n - c_n\rangle \to * \\ g &: \langle i - *;\ j_1 - c_1;\ \ldots;\ j_n - c_n\rangle \to * \end{aligned}$$

we will write the type instantiation as

$$g[f/i] : \langle j_1 - c_1;\ \ldots;\ j_n - c_n\rangle \to *$$

to mean the composition

$$\langle i : f;\ j_1 : j_1;\ \ldots;\ j_n : j_n\rangle\ .\ g$$

This generalizes easily to type-instantiation of values.

Now we drop the type and operations associated with functions (which are problematic due to the contravariance — dealing with that is tiresome). Thereafter, our two stories are linked together, because what we got is just a 2-category with (many) products. In analogy with **Cat** we will talk about kinds as categories, about types as functors and about values as transformations. Once we have functors let us make some adjoints!

The categorical story revisited

Let us look closer at some adjunctions in **Cat**. This is a repetition of the above toying with adjoints, but this time in more detail and more precisely. To simplify things, we may still identify the `*` category of basic types with the category **Set**.

Let $\Delta : \mathbf{Set} \to \texttt{<}i_1\ \texttt{-}\ \mathbf{Set}\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{-}\ \mathbf{Set}\texttt{>}$ be a functor given by $A \mapsto \texttt{<}i_1\ \texttt{:}\ A\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ A\texttt{>}$. This functor can be expressed inside our calculus in the following way.

$$\Delta \quad = \quad \texttt{<}i_1\ \texttt{: F\_ID(*);}\ \ldots\texttt{;}\ i_n\ \texttt{: F\_ID(*)>}$$

We've said that the right adjunction to Δ, the labeled product, provides the record datatype. What does it mean?

We will use the presentation of adjunctions based on factorizer (the transpose operation) which provides a co-unit, denoted ε_R and a factorizer, denoted `T_rfact`. These are bound by the following two equations.

$$\begin{array}{rcl}\texttt{T\_rfact}(\Delta, f, (u * \Delta)\ \texttt{.}\ (f' * \varepsilon_R(\Delta))) & = & u \\ (\texttt{T\_rfact}(\Delta, f, t) * \Delta)\ \texttt{.}\ (h * \varepsilon_R(\Delta)) & = & t\end{array}$$

We will define a projection based on the co-unit and a record operation based on the factorizer.

$$\begin{array}{rcl} j & = & h * \varepsilon_R(\Delta) * \texttt{<}j\texttt{>} \\ \texttt{\{}i\ \texttt{=}\ t\texttt{;}\ \ldots\texttt{\}} & = & \texttt{T\_rfact}(\Delta, f, \texttt{<}i\ \texttt{=}\ t\texttt{;}\ \ldots\texttt{>})\end{array}$$

which results in the same typing as before, where the labeled product (the right adjoint to Δ) is denoted using curly braces.

$$\frac{}{i \rhd \texttt{\{}i\ \texttt{:}\ f\texttt{;}\ \ldots\texttt{\}} \to f} \tag{30}$$

$$\frac{t_1 \rhd f \to h_1 \quad \cdots \quad t_n \rhd f \to h_n}{\texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}} \rhd f \to \texttt{\{}i_1\ \texttt{:}\ h_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ h_n\texttt{\}}} \tag{31}$$

When written using this notation, the adjunction equations from above look as follows.

$$\begin{array}{rcl}\texttt{\{}i\ \texttt{=}\ u\ \texttt{.}\ i\texttt{;}\ \ldots\texttt{\}} & = & u \\ \texttt{<}i_1\ \texttt{=}\ \texttt{\{}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ i_1\texttt{;}\ \ldots\texttt{>} & = & \texttt{<}i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{>}\end{array}$$

Now let us try to devise something similar to the functional datatype using the above functors. Let $B, B_1, \ldots, B_n$ be sets. We will first express in our calculus the candidate functors given by

$$\begin{array}{rcl}
A & \mapsto & \texttt{\{}j\ \texttt{:}\ A\texttt{;}\ i_1\ \texttt{:}\ B\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B\texttt{\}} \\
A & \mapsto & \texttt{\{}j\ \texttt{:}\ A\texttt{;}\ i_1\ \texttt{:}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B_n\texttt{\}} \\
A & \mapsto & \texttt{\{}j_1\ \texttt{:}\ A\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A\texttt{;}\ i\ \texttt{:}\ B\texttt{\}} \\
\texttt{<}j_1\ \texttt{:}\ A_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A_n\texttt{>} & \mapsto & \texttt{\{}j_1\ \texttt{:}\ A_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A_n\texttt{;}\ i\ \texttt{:}\ B\texttt{\}}
\end{array}$$

in the following way

$$\begin{array}{l}
\texttt{\{}j\ \texttt{: F_ID(*);}\ i_1\ \texttt{: <> .}\ B\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{: <> .}\ B\texttt{\}} \\
\texttt{\{}j\ \texttt{: F_ID(*);}\ i_1\ \texttt{: <> .}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{: <> .}\ B_n\texttt{\}} \\
\texttt{\{}j_1\ \texttt{: F_ID(*);}\ \ldots\texttt{;}\ j_n\ \texttt{: F_ID(*);}\ i\ \texttt{: <> .}\ B\texttt{\}} \\
\texttt{\{}j_1\ \texttt{:}\ j_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ j_n\texttt{;}\ i\ \texttt{: <> .}\ B\texttt{\}}
\end{array}$$

where $B, B_1, \ldots, B_n : \texttt{<>} \to \texttt{*}$.

Let us start with the second functor. The right adjoint to the functor

$$\texttt{\{}j\ \texttt{: F\_ID(*);}\ i_1\ \texttt{: <> .}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{: <> .}\ B_n\texttt{\}}$$

(when composed with a functor h) will be denoted by

$$\texttt{\~{}}i_1\texttt{:}B_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}B_n\ \texttt{-}j\texttt{->}\ h$$

The operations obtained from the adjunction will be written and typed as follows, where $g = \texttt{\~{}}i_1\texttt{:}B_1\ \ldots\ \texttt{\~{}}i_n\texttt{:}B_n\ \texttt{-}j\texttt{->}\ h$.

$$\frac{}{\texttt{\~{}}j\texttt{\~{}} \triangleright \texttt{\{}j\ \texttt{:}\ g\texttt{;}\ i_1\ \texttt{:}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B_n\texttt{\}} \to h} \tag{32}$$

$$\frac{t \triangleright \texttt{\{}j\ \texttt{:}\ f\texttt{;}\ i_1\ \texttt{:}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B_n\texttt{\}} \to h}{\texttt{fun \~{}}i_1\ \ldots\ \texttt{\~{}}i_n\ \texttt{-}j\texttt{->}\ t \triangleright f \to g} \tag{33}$$

The equations we get from the definition of adjunction are the following.

$$\begin{array}{rcl}\texttt{fun \char`~}i_1\ \ldots\ \texttt{\char`~}i_n\ \texttt{-}j\texttt{->}\ (\texttt{\{}j\ \texttt{=}\ j\ \texttt{.}\ u\texttt{;}\ i_1\ \texttt{=}\ i_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{\char`~}j\texttt{\char`~}) & = & u \\ \texttt{\{}j\ \texttt{=}\ (j\ \texttt{.}\ \texttt{fun \char`~}i_1\ \ldots\ \texttt{\char`~}i_n\ \texttt{-}j\texttt{->}\ t)\texttt{;}\ i_1\ \texttt{=}\ i_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ \texttt{\char`~}j\texttt{\char`~} & = & t\end{array}$$

The ordinary operations of application and abstraction may now be recovered in the following way. Let $t : \texttt{\char`~}i_1\texttt{:}B_1\ \ldots\ \texttt{\char`~}i_n\texttt{:}B_n\ \texttt{-}j\texttt{->}\ h$.

$$\begin{array}{rcl}t\ \texttt{\char`~}i_1\texttt{:}t_1\ \ldots\ \texttt{\char`~}i_n\texttt{:}t_n & = & \texttt{\{}j\ \texttt{=}\ t\texttt{;}\ i_1\ \texttt{=}\ t_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{=}\ t_n\texttt{\}}\ \texttt{.}\ \texttt{\char`~}j\texttt{\char`~} \\ \texttt{fun \char`~}j_1\ \ldots\ \texttt{\char`~}j_k\ \texttt{->}\ t & = & \texttt{fun \char`~}j_1\ \ldots\ \texttt{\char`~}j_k\ \texttt{-}j\texttt{->} \\ & & \quad \texttt{\{}i_1\ \texttt{=}\ j\ \texttt{.}\ i_1\texttt{;}\ \ldots\texttt{;}\ j_1\ \texttt{=}\ j_1\texttt{;}\ \ldots\texttt{\}}\ \texttt{.}\ t\end{array}$$

Theorem. *If the j label is set constant, the theory of datatype*

$$\texttt{\char`~}i_1\texttt{:}B_1\ \ldots\ \texttt{\char`~}i_n\texttt{:}B_n\ \texttt{-}j\texttt{->}\ h$$

is equivalent to the theory of datatype

$$\texttt{\{}i_1\ \texttt{:}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B_n\texttt{\}}\ \texttt{->}\ h$$

When the theorem is strictly and categorically formulated the proof amounts to showing an isomorphism between

$$\texttt{\{}j_1\ \texttt{:}\ A\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A\texttt{;}\ i\ \texttt{:}\ B\texttt{\}}$$

and

$$\texttt{\{}j = \texttt{\{}j_1\ \texttt{:}\ A\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A\texttt{\}}\texttt{;}\ i\ \texttt{:}\ B\texttt{\}}$$

We return to the candidate functors. As the first candidate functor is less general than the second, we see that it corresponds to a labeled function space with all the arguments of the same type.

Notice that the labeled exponent based on the second candidate have all the "context", the free variables that are to stay in, stored at one index j. On the other hand the ordinary abstraction operation recovered above has the following informal typing, where the all the free variables of the argument are listed together (here $g = \texttt{\char`~}i_1\texttt{:}B_1\ \ldots\ \texttt{\char`~}i_n\texttt{:}B_n\ \texttt{->}\ h$).

$$\frac{t \rhd \texttt{\{}j_1\ \texttt{:}\ A_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A_n\texttt{;}\ i_1\ \texttt{:}\ B_1\texttt{;}\ \ldots\texttt{;}\ i_n\ \texttt{:}\ B_n\texttt{\}} \to h}{\texttt{fun \char`~}i_1\ \ldots\ \texttt{\char`~}i_n\ \texttt{->}\ t \rhd \texttt{\{}j_1\ \texttt{:}\ A_1\texttt{;}\ \ldots\texttt{;}\ j_n\ \texttt{:}\ A_n\texttt{\}} \to g} \tag{34}$$

Can we get such a typing directly from an adjunction? Perhaps we can obtain such a behavior by analyzing the third candidate and then appropriately merging the second and the third candidate functors.

Let us try the third candidate functor. One might expect that it will result in a labeled function space with one argument and many results. Is it so? Let us call the third candidate functor g and the right adjoint $\texttt{F\_rad}(g)$. The right adjunction gives us the co-unit and the factorizer, where $t : f \;\texttt{.}\; g \to h$.

$$\begin{aligned}\varepsilon_R(g) &: \texttt{F\_rad}(g) \;\texttt{.}\; g \to \texttt{F\_ID}(e)\\ \texttt{T\_rfact}(g, f, t) &: f \to h \;\texttt{.}\; \texttt{F\_rad}(g)\end{aligned}$$

If we multiply the co-unit by h and abbreviate $h \;\texttt{.}\; \texttt{F\_rad}(g)$ to $r(h)$ we get

$$\begin{aligned}h * \varepsilon_R(g) &: \texttt{\{}j_1 \;\texttt{:}\; r(h)\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{:}\; r(h)\texttt{;}\; i \;\texttt{:}\; \texttt{<>} \;\texttt{.}\; B\texttt{\}} \to h\\ \texttt{T\_rfact}(g, f, t) &: f \to r(h)\end{aligned}$$

where $t : \texttt{\{}j_1 \;\texttt{:}\; f\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{:}\; f\texttt{;}\; i \;\texttt{:}\; \texttt{<>} \;\texttt{.}\; B\texttt{\}} \to h$. These typings show that indeed we have "functions" with single arguments, but the results seem to be single as well. What is numerous are free variables but these all have to have the same type.

We have these equations

$$\begin{aligned}\texttt{T\_rfact}(g, f, (u * g) \;\texttt{.}\; (f' * \varepsilon_R(g))) &= u\\ (\texttt{T\_rfact}(g, f, t) * g) \;\texttt{.}\; (h * \varepsilon_R(g)) &= t\end{aligned}$$

The second equation

$$\begin{aligned}&\texttt{\{}j_1 \;\texttt{=}\; \texttt{T\_rfact}(g, f, t)\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{=}\; \texttt{T\_rfact}(g, f, t)\texttt{;}\\ &\;\; i \;\texttt{=}\; \texttt{T\_ID(<>} \;\texttt{.}\; B\texttt{)\}} \;\texttt{.}\; (h * \varepsilon_R(g)) \;=\; t\end{aligned}$$

says that to perform an application we have to gather n identical "functions" and a single argument. As a result the "context" is multiplied to fit the many identically typed free variables of the function body and the argument is composed with the body.

So what we can obtain here is, informally,

$$\frac{t \triangleright \texttt{\{}j_1 \;\texttt{:}\; A\texttt{;}\; \ldots\texttt{;}\; j_n \;\texttt{:}\; A\texttt{;}\; i \;\texttt{:}\; B\texttt{\}} \to h}{\texttt{fun \~{}}i \;\texttt{->}\; t \triangleright A \to (\texttt{\~{}}i\texttt{:}B \;\texttt{->}\; h)} \tag{35}$$

which seems promising, if not for the identical typing of the free variables. Perhaps the last candidate functor will be general enough to allow arbitrary free variables.

We proceed with the last, the fourth candidate functor. Let us call it g and the right adjoint to it $\texttt{F\_rad}(g)$. The right adjunction gives us the co-unit and the factorizer, where $t : f \texttt{ . } g \to h$.

$$\begin{array}{rcl} \varepsilon_R(g) & : & \texttt{F\_rad}(g) \texttt{ . } g \to \texttt{F\_ID}(e) \\ \texttt{T\_rfact}(g, f, t) & : & f \to h \texttt{ . } \texttt{F\_rad}(g) \end{array}$$

If we multiply the co-unit by h and abbreviate $h \texttt{ . } \texttt{F\_rad}(g)$ to $r(h)$ we get

$$\begin{array}{rcl} h * \varepsilon_R(g) & : & \texttt{\{}j_1 \texttt{ : } r(h) \texttt{ . } j_1\texttt{; } \ldots\texttt{; } j_n \texttt{ : } r(h) \texttt{ . } j_n\texttt{; } i \texttt{ : <> . } B\texttt{\}} \to h \\ \texttt{T\_rfact}(g, f, t) & : & f \to r(h) \end{array}$$

where $t : \texttt{\{}j_1 \texttt{ : } f \texttt{ . } j_1\texttt{; } \ldots\texttt{; } j_n \texttt{ : } f \texttt{ . } j_n\texttt{; } i \texttt{ : <> . } B\texttt{\}} \to h$. Again we got "functions" with single arguments and single results. But this time the free variables may have different types, which is good.

We have these equations

$$\begin{array}{rcl} \texttt{T\_rfact}(g, f, (u * g) \texttt{ . } (f' * \varepsilon_R(g))) & = & u \\ (\texttt{T\_rfact}(g, f, t) * g) \texttt{ . } (h * \varepsilon_R(g)) & = & t \end{array}$$

The second equation

$$\begin{array}{l} \texttt{\{}j_1 \texttt{ = } \texttt{T\_rfact}(g, f, t) * \texttt{<}j_1\texttt{>; } \ldots\texttt{; } j_n \texttt{ = } \texttt{T\_rfact}(g, f, t) * \texttt{<}j_n\texttt{>;} \\ \quad i \texttt{ = T\_ID(<> . } B\texttt{)\} . } (h * \varepsilon_R(g)) \; = \; t \end{array}$$

says that to perform an application we have to gather all projections of a single "function" and provide a single argument.

The abstraction now is typed roughly as follows

$$\frac{t \rhd \texttt{\{}j_1 \texttt{ : } A_1\texttt{; } \ldots\texttt{; } j_n \texttt{ : } A_n\texttt{; } i \texttt{ : } B\texttt{\}} \to h}{\texttt{fun \~{}}i \texttt{ -> } t \rhd \texttt{<}j_1 \texttt{ : } A_1\texttt{; } \ldots\texttt{; } j_n \texttt{ : } A_n\texttt{>} \to (\texttt{\~{}}i\texttt{:}B \texttt{ -> } h)} \tag{36}$$

and this is still wrong, because the domain of the abstraction is a wrong kind of a product. In fact, the abstraction itself is of a product kind, which means it may not be a first-class value in our language.

There are many other strange phenomena with this datatype. For example, the various projections of the abstraction are of the base kind and each one of them has only one free variable and moreover this variable is anonymous!

Appendix: Basic axioms

Equations of functors.

$$
\begin{aligned}
f \texttt{ . F\_ID}(d) &= f && (1)\\
\texttt{F\_ID}(c) \texttt{ . } g &= g && (2)\\
f \texttt{ . } (g_1 \texttt{ . } g_2) &= (f \texttt{ . } g_1) \texttt{ . } g_2 && (3)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{<}i \texttt{ : } f\texttt{; } \ldots\texttt{> . } i &= f && (4)\\
\texttt{<}i_1 \texttt{ : } f \texttt{ . } i_1\texttt{; } \ldots\texttt{; } i_n \texttt{ : } f \texttt{ . } i_n\texttt{>} &= f && (5)
\end{aligned}
$$

Equations of transformations. Notice the symmetries.

$$
\begin{aligned}
t \texttt{ . T\_id}(g) &= t && (6)\\
\texttt{T\_id}(f) \texttt{ . } t &= t && (7)\\
t \texttt{ . } (u_1 \texttt{ . } u_2) &= (t \texttt{ . } u_1) \texttt{ . } u_2 && (8)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{T\_ID}(c) &= \texttt{T\_id(F\_ID}(c)) && (9)\\
\texttt{T\_id}(f) * \texttt{T\_id}(g) &= \texttt{T\_id}(f \texttt{ . } g) && (10)\\
(t_1 \texttt{ . } t_2) * (u_1 \texttt{ . } u_2) &= (t_1 * u_1) \texttt{ . } (t_2 * u_2) && (11)
\end{aligned}
$$

$$
\begin{aligned}
t * \texttt{T\_ID}(d) &= t && (12)\\
\texttt{T\_ID}(c) * u &= u && (13)\\
t * (u_1 * u_2) &= (t * u_1) * u_2 && (14)
\end{aligned}
$$

$$
\begin{aligned}
\texttt{<}i \texttt{ = } t\texttt{; } \ldots\texttt{>} * \texttt{<}i\texttt{>} &= t && (15)\\
\texttt{<}i_1 \texttt{ = } t * \texttt{<}i_1\texttt{>; } \ldots\texttt{; } i_n \texttt{ = } t * \texttt{<}i_n\texttt{>>} &= t && (16)
\end{aligned}
$$

$$
\texttt{T\_id}(i) = \texttt{<}i\texttt{>} \qquad (17)
$$

All the rest follows from particular adjunctions for particular datatypes.