Pattern Matching vs. Elimination Rules
A Case Study in LEGO

Andreas Abel*
abel@informatik.uni-muenchen.de
http://www.tcs.informatik.uni-muenchen.de/~abel/
Department of Computer Science
University of Munich

September 30, 1999

1 Introduction

In Type Theory there are two paradigms of how to obtain functions over recursive types: Either by recursion resp. elimination rules or as the fixed-point of a structural recursive definition. In [Gim94] Eduardo Giménez has shown that both methods are equivalent. On the one hand one can codify the elimiation rules as structural recursive functions. On the other hand he describes a method of how from a function f defined as a fixed-point one obtains an extensionally equal function f' using only elimination rules.

In the following we exercise his method on a small example. Doing this we want to demonstrate that pattern matching definitons are often much more straight forward and understandable than elimination style definitions are. (In my case, it took me a quater of an hour to do the pattern matching proof, but two full days to transform it into a elimination proof. *Writing down this paper I then realized that using a small trick there is actually a very simple elimination style formulation of my example. This destroyed my aim 1a ("more straight forward") but emphasized aim 1b ("understandable").*) The second aim of this paper is to show how to do it in *LEGO*. Therefore we enclose the full LEGO code after each section.

*Member of the PhD Programme Logics in Computer Science

2 The Task

Im my diploma thesis [Abe99] I showed that all good values in the foetus type system are accessible with regard to a structural ordering on the values. In the following we introduce an untyped and very cut-down version of the set of values and rules determining an ordering on it and want to show that one additional rule is admissible.

The set V of values is generated by the introduction rules

$$\begin{array}{ll}
\text{(vunit)} & () \in V \\
\text{(vpair)} & \forall v, w \in V.\ (v, w) \in V \\
\text{(vfold)} & \forall v \in V.\ \mathrm{fold}\, v \in V
\end{array}$$

In LEGO we define V as inductive set. The constructors correspond to the introduction rules we have given above. Indeed in LEGO both are seen as one.

```
Module admissible Import lib_bool;
[PROP=Type];
[Set=Type];

Inductive [V:Set]
Constructors
        [vunit: V]
        [vpair: V->V->V]
        [vfold: V->V];
```

We define the ordering relations $<, \leq \subset V \times V$ by the following rules:

$$\text{(lerefl)} \quad v \leq v$$

$$\text{(lelt)} \quad \frac{w < v}{w \leq v}$$

$$\text{(lefst)} \quad \frac{w \leq v}{w \leq (v, v')} \qquad \text{(ltfst)} \quad \frac{w < v}{w < (v, v')}$$

$$\text{(lefold)} \quad \frac{w \leq v}{w < \mathrm{fold}\, v} \qquad \text{(ltfold)} \quad \frac{w < v}{w < \mathrm{fold}\, v}$$

The intuition behind these rules is: Pairing on the right side preserves the order, adding a constructor (fold) on the right side increases the order.

In LEGO for simplicity we introduce just one relation $L \subseteq \mathbb{B} \times V \times V$ and will represent "$<$" by L_{true} and "$\leq$" by L_{false}.

```
Inductive [L: bool->V->V->PROP]
Constructors
        [lerefl: {v:V}L false v v]
        [lelt: {w,v|V}(L true w v)->L false w v]
        [l_fst: {b|bool}{w,v|V}(L b w v)->{v':V}L b w (vpair v v')]
        [l_fold: {b|bool}{w,v|V}(L b w v)->L true w (vfold v)];
```

What we now want to show is that liquidating a pair on the left side also preserves the order. Therefore we have to prove that the rules

$$(\mathrm{lefst}') \quad \frac{(w, w') \leq v}{w \leq v} \qquad (\mathrm{ltfst}') \quad \frac{(w, w') < v}{w < v}$$

are *admissible*, i.e. that whenever we can derive the premise there exists a deriviation of the conclusion. (In contrast to *derivable* what means that we can derive the conclusion *from* the premise by applying the basic rules.)

3 The pattern matching proof

The proof is done by induction over the deriviation of the premise:

(lerefl) The premise is of form $(w, w') \leq (w, w')$. Therefore the conclusion we must prove is $w \leq (w, w')$. This is done by (lerefl) followed by (lefst).

(lelt) The premise has been derived by

$$\frac{(w, w') < v}{(w, w') \leq v}$$

We must show $w \leq v$. By applying the induction hypothesis on $(w, w') < v$ we have $w < v$ and we finish this case by (lelt).

(lefst) The last deriviation step of the premise was

$$\frac{(w, w') \leq v}{(w, w') \leq (v, v')}$$

Again by induction hypothesis we have to show

$$\frac{w \le v}{w \le (v, v')}$$

which we get immediately by (lefst).

(ltfst) analogously

(lefold) Here we have obtained the premise by

$$\frac{(w, w') \le v}{(w, w') \le \mathrm{fold}\, v}$$

We complete the proof by induction hypothesis and (lefold).

(ltfold) analogously □

This proof can be coded directly via pattern matching in LEGO as follows. It is a function

$$\text{l–fst}' : \forall b{:}\mathbb{B}, w, w', v{:}V.\ (w, w')L_b v \to wL_b v$$

that takes a proof tree of $(w, w')L_b v$ and computes a proof tree of $wL_b v$.

```
$[l_fst': {b|bool}{w,w',v|V}(L b (vpair w w') v)->L b w v];
[[b:bool][w,w',v,v':V][h:L b (vpair w w') v]
 [h1:L true (vpair w w') v]
        l_fst' (lerefl (vpair w w')) ==> l_fst (lerefl w) w'
||      l_fst' (lelt h1) ==> lelt (l_fst' h1)
||      l_fst' (l_fst h v') ==> l_fst (l_fst' h) v'
||      l_fst' (l_fold h) ==> l_fold (l_fst' h)
];
```

4 The elimination proof a la Gimenéz

Following Gimenéz approach we first have to define a new inductive type

$$L^2 : \forall b{:}\mathbb{B}, w, w', v{:}V.\ (w, w')L_b v \to \mathrm{Set}$$

indexed over the deriviation trees of $(w, w')L_b v$ that "stores" the derviation tree of $wL_b v$. Thus we can define the function

$$\text{l–fst}'\mathrm{body} : \forall b{:}\mathbb{B}, w, w', v{:}V, p{:}(w, w')L_b v.\ L^2 p \to wL_b v$$

which is the "body" of l-fst$'$ with the modification that the recursive calls are replaced by their results that now are taken out of the structure L^2. The last element is a function

$$\mathrm{genL}^2 : \forall b{:}\mathbb{B}, w, w', v{:}V, p{:}(w,w')L_b v.\ L^2 p$$

that generates the structure L^2 using l-fst$'$body. The desired function l-fst$'$ is then just the composition of both:

$$\text{l–fst}' = \lambda p{:}(w,w')L_b v.\ \text{l–fst}'\text{body}\ p\ (\mathrm{genL}^2\, p)$$

Trying to implement this in LEGO raises one problem: It is not possible to do L-induction on a goal of the form

$$(w,w')L_b v \to P$$

We have to transform it to the form

$$uL_b v \to u = (w,w') \to P$$

instead. (To realize this caused me a lot of trouble and required several unsuccessfull tries.) As one consequence we have to adapt the type of L^2 and the auxiliary functions.

```
Inductive [L2: {b|bool}{u,v|V}(L b u v)->
               {w,w'|V}(Eq u (vpair w w'))->Set]
Constructors
[l2refl: {u,w,w'|V}{e:Eq u (vpair w w')}
                L2 (lerefl u) e]
[l2lt: {u,v|V}{p:L true u v}{w,w'|V}{e:Eq u (vpair w w')}
                 (L true w v)->(L2 p e)->L2 (lelt p) e]
[l2fst:  {b|bool}{u,v|V}{p:L b u v}{w,w'|V}{e:Eq u (vpair w w')}
                (L b w v)->(L2 p e)->
                {v':V}L2 (l_fst p v') e]
[l2fold: {b|bool}{u,v|V}{p:L b u v}{w,w'|V}{e:Eq u (vpair w w')}
                (L b w v)->(L2 p e)->L2 (l_fold p) e];
```

Instead of directly writing down the lambda term for l-fst$'$body we use the interactive mode of the LEGO proof assistant. The function is defined by induction (recursion) over $H : L^2$.

```
Goal l_fst'body: {b|bool}{u,v|V}{p|L b u v}
        {w,w'|V}{e|Eq u (vpair w w')}(L2 p e)->L b w v;
Intros;
Refine L2_elim [b|bool][u,v|V][p|L b u v]
        [w,w'|V][e|Eq u (vpair w w')][H:L2 p e]L b w v;
```

```
(* l2refl: *) Intros; Qrepl e1; Refine l_fst; Refine lerefl;
(* l2lt:   *) Intros; Refine lelt; Immed;
(* l2fst:  *) Intros; Refine l_fst; Immed;
(* l2fold: *) Intros; Refine l_fold; Immed;
Immed;
Save;
```

The function genL^2 is defined by recursion over $p : L$ and makes use of $\text{l-fst}'\text{body}$.

```
Goal genL2: {b|bool}{u,v|V}{p:L b u v}
        {w,w'|V}{e:Eq u (vpair w w')}L2 p e;
Intros b u v p;
Refine L_elim [b|bool][u,v|V][p:L b u v]
        {w,w'|V}{e:Eq u (vpair w w')}L2 p e;
(* lerefl: *) Intros; Refine l2refl;
(* lelt:   *) Intros; Refine l2lt;   Refine l_fst'body; Immed;
(* l_fst:  *) Intros; Refine l2fst;  Refine l_fst'body; Immed;
(* l_fold: *) Intros; Refine l2fold; Refine l_fst'body; Immed;
Save;
```

Since $\text{l-fst}'$ is just the composition of $\text{l-fst}'\text{body}$ and genL^2 we write it directly as a lambda-term.

```
[l_fst' : {b|bool}{w,w',v|V}(L b (vpair w w') v)->L b w v
        = [b|bool][w,w',v|V][H:L b (vpair w w') v]
          l_fst'body (genL2 H (Eq_refl (vpair w w')))];
```

5 A direct elimination proof

Since $\text{l-fst}'$ calls itself only with *direct* subcomponents of the input parameter, it is (nearly) primitive recursive and therefore should need no additional inductive types. But again we have the problem that the L-elimination rule is not applicable on

$$(w, w')L_b v \to P$$

Using the Eq-trick however we can code a primitive recursive auxiliary function

$$\text{l–fst}'\text{aux} : uL_b v \to u = (w, w') \to wL_b v$$

and then have $\text{l–fst}' = \lambda p : (w, w')L_b v.\ \text{l–fst}'\text{aux}\ p\ e$ with $e : (w, w') = (w, w')$.

```
Goal l_fst'aux: {b|bool}{u,v|V}{p:L b u v}
                {w,w'|V}{e:Eq u (vpair w w')}L b w v;
```

```
Intros b u v p;
Refine L_elim   [b|bool][u,v|V][p:L b u v]
                {w,w'|V}{e:Eq u (vpair w w')}L b w v;
(* lerefl: *) Intros; Qrepl e; Refine l_fst; Refine lerefl;
(* lelt:   *) Intros; Refine lelt;   Refine    x3_ih; Immed;
(* l_fst:  *) Intros; Refine l_fst;  Refine    x2_ih; Immed;
(* l_fold: *) Intros; Refine l_fold; Refine +1 x1_ih; Immed;
Immed;
Save;

[l_fst' : {b|bool}{w,w',v|V}(L b (vpair w w') v)->L b w v
        = [b|bool][w,w',v|V][H:L b (vpair w w') v]
          l_fst'aux H (Eq_refl (vpair w w'))];
```

References

[Abe99] Andreas Abel. A semantic analysis of structural recursion. Master's thesis, Ludwig-Maximilians-University Munich, 1999. http://www.informatik.uni-muenchen.de/~abel/publications/.

[Gim94] Eduardo Gimenéz. Codifying guarded definitions with recursive schemes. Technical Report 95-07, Ecole Normale Supérieure de Lyon, December 1994.