IPL by Examples

Zdzisław Spławski
Technical University of Wrocław*

October 18, 1993

I have introduced Inferential Programming Language and Logic (IPL) in my doctoral thesis "Proof-Theoretic Approach to Inductive Definitions in ML-like Programming Language versus Second-Order Lambda Calculus". In IPL I try to combine the merits of two main paradigmes of polymorphic typing: parametric quantifier-free polymorphism, as in Standard ML, and Girard and Reynolds's impredicative quantificational system F. IPL and SML share type inference mechanism, however all programs written in IPL terminate under every reduction strategy (strong normalization property). This entails decidability of (intensional) equality and possibility of making proofs by symbolic execution of programs. IPL is built over one predefined data type "$\rightarrow$", but new inductively defined data types may be added (in much the same way as in SML), provided they fulfill the positivity condition. In contrast with SML the system responds to a data type declaration not only with types of constructors but also with type of the eliminator (iterator and recursor) together with its equational definition. All function definitions are explicit, recursion being provided by eliminators (in that respect IPL is similar to Martin-Löf's type theory, though there are no dependent types in IPL). This is dualized to coinductive definitions of data types with coiterators and corecursors. I do not see any *useful* data type or program expressible in F, but not expressible in IPL.

A data type comprises a type and some characteristic operations. A data type may be parameterized, that is, defined as a function on types, in which case all the operations are parameterized. The analogies between formal systems and IPL are summarized below.

logic, formal system	$\sim$	functional language
logical connective	$\sim$	(co)data type
introduction rules	$\sim$	constructors (introducers)
elimination rules	$\sim$	eliminators (destructors)
constructive proof	$\sim$	program, i.e. polymorphic function
proposition	$\sim$	type of a program
normalization	$\sim$	computation
equivalences of proofs	$\sim$	convertibility

Inductive data types are defined by the set of constructors with one generated eliminator; coinductive data types are defined by the set of destructors with one generated introducer. In most other approaches constructors (destructors) and eliminators (introducers) must be defined separately.

IPL is a functional language based on primitive recursion and corecursion and higher order functions, hence it can be viewed from another angle as an extension of Gödel's system T to arbitrary polymorphic (co)inductively defined data types.

Experimental implementation of IPL has been programmed in Standard ML.

Besides function space `->` there are some other predefined data types, namely empty type {} with eliminator `case0`, singleton type `UNIT` with constructor `()` and eliminator `case1`, product type `*` with infix constructor denoted somehow unusually by a comma and two destructors `fst` and `snd`, and union type `+` with two constructors `Inl, Inr` and eliminator `when`. They can be defined in IPL, but product (union) type is used in types of generated recursors (resp. corecursors). A function space type constructor `->` associates to the right and binds weaker than a union type constructor `+` which is weaker than a

*Author's address: Centre of Informatics, Technical University of Wrocław, Wybrzeże Wyspiańskiego 27, 50–370 Wrocław, Poland. Electronic mail: splawski@plwrtu11.bitnet product type constructor `*`. Union and product type constructors associate to the left. In responses, for readability, the IPL system uses more parentheses than required, but fewer than SML does.

There is also the built-in equality predicate

```
= :  'a -> 'a -> BOOL
```

thus we had to predeclare the data type `BOOL` with two constructors `True, False` and eliminator `if` *term* `then` *term* `else` *term*. Identifiers beginning with an apostrophe, e.g. `'a, 'b, 'c` ..., denote type variables. Names of (co)iterators and (co)recursors are generated from (co)datatype constructors by adding an underscore at the beginning and `it` (for iterators), `rec` (for recursors), `ci` (for coiterators) or `cr` (for corecursors) at the end of (co)datata type constructor.

The system prompts a user with `+` at the beginning of a line. Continuation lines are preceded by `=`. System responses are not preceded by any sign.

The syntax of IPL resembles the syntax of Standard ML. Below IPL is presented by examples.

Value declarations are similar to declarations in SML.

```
+ val f1 = fn p => fn q => fn r => p r;
val f1 : ('a -> 'b) -> 'c -> 'a -> 'b
```

However, thanks to strong normalization, functions are evaluated to normal forms, not to weak head normal forms as in most functional programming languages, and their values are displayed:

```
+ f1;
fn z y => z : ('a -> 'b) -> 'c -> 'a -> 'b
```

We see, that bound variables are different from those used in the definition of `f1`. This is because the internal representation is based on de Bruijn nameless terms and the names of bound variables are generated by the system. η-reduction is also performed and, following the convention of λ-calculus, one keyword `fn` can be used for more than one abstraction (which is not allowed in SML):

```
+ (fn x y z => x z) = it;
True : BOOL
```

In IPL (unlike SML) equality between terms of arbitrary types is decidable. The identifier `it` has the same meaning as in SML. After each evaluation of an expression, the IPL system binds the value of the expression to `it`. Below we define two standard combinators `s, k`, and show that `skk` defines identity function.

```
+ val s = fn f g x => f x (g x);
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c
+ val k = fn x y => x;
val k : 'a -> 'b -> 'a
+ s k k;
fn z => z : 'a -> 'a
```

We can also use local declarations in expressions, e.g.

```
+ val i = let val s = fn f g x => f x (g x);
=             val k = fn x y => x;
=         in s k k end;
val i : 'a -> 'a
+ i;
fn z => z : 'a -> 'a
```

Inductive data types are defined, like in SML, by enumerating their constructors (which encode introduction rules), with types of arguments. Natural numbers can be defined inductively as follows:

```
+ datatype NAT = Suc from NAT | 0;
```

Object constructors are in curried form (like in Miranda) and there is a separator `from` instead of `of` used in SML. Omitting `from` and the following type expression has the same meaning as in SML — non-functional constructor is created. System output looks as follows:

```
datatype NAT
con  Suc : NAT -> NAT
con  0 : NAT
iter _NATit : NAT -> ('a -> 'a) -> 'a -> 'a
comp _NATit (Suc z) = fn y x => y (_NATit z y x)
comp _NATit 0 = fn z y => y
rec  _NATrec : NAT -> (NAT * 'a -> 'a) -> 'a -> 'a
comp _NATrec (Suc z) = fn y x => y (z , _NATrec z y x)
comp _NATrec 0 = fn z y => y
```

System output contains both iterator and recursor. In presence of recursor, iterator is of course redundant. I have included it for comparison, because we implemented IPL to have a tool for experimentation. Besides in normal forms only constructors and eliminators can appear, hence normal forms are often simpler if we can use iterators as eliminators, not as defined functions. We may derive reduction rules by reading the generated equations from left to right.

Below we have equational specification of less or equal function `le`:

$$\begin{aligned}\texttt{le 0}\ v &= \texttt{True}\\ \texttt{le (Suc}\ u\texttt{) 0} &= \texttt{False}\\ \texttt{le (Suc}\ u\texttt{) (Suc}\ v\texttt{)} &= \texttt{le}\ u\ v\end{aligned}$$

We may define it in IPL:

```
+ val le = fn u => _NATit u (fn y v => _NATrec v (fn x => y (fst x)) False)
=                           (fn v => True) ;
val le : NAT -> NAT -> BOOL
```

and verify:

```
+ ( fn v => le 0 v ) = ( fn v => True );
True : BOOL
+ ( fn u => le (Suc u) 0 )  = ( fn u => False );
True : BOOL
+ ( fn u v => le (Suc u) (Suc v) ) = ( fn u v => le u v );
True : BOOL
```

Free variables of the specification could have been bound using one external abstraction, e.g.

```
+ fn u v => ( le (Suc u) (Suc v) ) = ( le u v );
fn z y => True : NAT -> NAT -> BOOL
```

As the next example we define the data type of polymorphic lists:

```
+ datatype LIST 'a = Cons from 'a (LIST 'a) | Nil;
datatype LIST 'a
con  Cons : 'a -> (LIST 'a) -> LIST 'a
con  Nil : LIST 'a
iter _LISTit : (LIST 'a) -> ('a -> 'b -> 'b) -> 'b -> 'b
comp _LISTit (Cons z y) = fn x w => x z (_LISTit y x w)
comp _LISTit Nil = fn z y => y
rec  _LISTrec : (LIST 'a) -> ('a -> (LIST 'a) * 'b -> 'b) -> 'b -> 'b
comp _LISTrec (Cons z y) = fn x w => x z (y , _LISTrec y x w)
comp _LISTrec Nil = fn z y => y
```

Both type and term constructors are written in prefix form and both can be curried. This is so, because I have an algorithm translating IPL with (co)iterators to F and I wanted the image in F to be as simple as possible. This algorithm has not been implemented, but it is possible to extend IPL to display this translation using Lego notation, say. Similarly I provided an algorithm translating IPL with (co)recursors to some extension of F.

Now we want to define head and tail functions. Tail is unproblematic:

```
+ val tl = fn xs => _LISTrec xs (fn h p => fst p) Nil;
val tl : (LIST 'a) -> LIST 'a
```

Before moving further we describe the built-in union data type, using IPL notation (certainly this definitions is illegal as it stands).

```
+ datatype  'a + 'b = Inl from 'a | Inr from 'b;
datatype 'a + 'b
con  Inl : 'a -> 'a + 'b
con  Inr : 'b -> 'a + 'b
iter when : ('a + 'b) -> ('a -> 'c) -> ('b -> 'c) -> 'c
comp when (Inl z) = fn y x => y z
comp when (Inr z) = fn y x => x z
```

Head applied to an empty list should rise an exception. There are no exceptions in IPL and we define head as a function from `LIST 'a` to a union of `'a` and some other unit type, which we interpret as head exception.

```
+ datatype HDexception = Hd;
datatype HDexception
con  Hd : HDexception
iter _HDexceptionit : HDexception -> 'a -> 'a
comp _HDexceptionit Hd = fn z => z
rec  _HDexceptionrec = _HDexceptionit
```

For non-(co)inductive data types (co)recursors are equal to (co)iterators, hence `_HDexceptionrec = _HDexceptionit`.

```
+ val hd = fn xs => _LISTit xs (fn h t => Inl h) (Inr Hd);
val hd : (LIST 'a) -> 'a + HDexception
```

Append function concatenates two lists.

```
+ val append = fn xs ys => _LISTit xs Cons ys;
val append : (LIST 'a) -> (LIST 'a) -> LIST 'a
```

In the next example we define the quicksort function. A list of elements to be sorted is transformed into a binary search tree, from which a sorted list is obtained by inorder traversal.

```
+ datatype BT 'a = Node from 'a (BT 'a) (BT 'a) | Tip;
datatype BT 'a
con  Node : 'a -> (BT 'a) -> (BT 'a) -> BT 'a
con  Tip : BT 'a
iter _BTit : (BT 'a) -> ('a -> 'b -> 'b -> 'b) -> 'b -> 'b
comp _BTit (Node z y x) = fn w v => w z (_BTit y w v) (_BTit x w v)
comp _BTit Tip = fn z y => y
rec  _BTrec : (BT 'a) -> ('a -> (BT 'a) * 'b -> (BT 'a) * 'b -> 'b) -> 'b -> 'b
comp _BTrec (Node z y x) = fn w v => w z (y , _BTrec y w v) (x , _BTrec x w v)
comp _BTrec Tip = fn z y => y
```

The function `toBST` adds one element to a binary search tree.

```
+ val toBST = fn x t => _BTrec t (fn a lp rp =>
=                                    if le x a then Node a (snd lp) (fst rp)
=                                              else Node a (fst lp) (snd rp)
=                                ) (Node x Tip Tip);
val toBST : NAT -> (BT NAT) -> BT NAT
+ val listToBST = fn xs => _LISTit xs toBST Tip;
val listToBST : (LIST NAT) -> BT NAT
+ val inorder = fn t => _BTit t (fn a l r => append l (Cons a r)) Nil;
val inorder : (BT 'a) -> LIST 'a
+ val quicksort = fn xs => inorder (listToBST xs);
val quicksort : (LIST NAT) -> LIST NAT
+ val one = Suc 0;
val one : NAT
+ val two = Suc one;
val two : NAT
+ val three = Suc two;
val three : NAT
+ val l = Cons two (Cons one (Cons three (Cons 0 Nil)));
val l : LIST NAT
+ quicksort l;
Cons 0 (Cons (Suc 0) (Cons (Suc (Suc 0)) (Cons (Suc (Suc (Suc 0))) Nil))) : LIST NAT
```

It is easy to check that this method has O($n \log n$) complexity. IPL uses lazy evaluation (though in view of strong normalization any reduction strategy could have been used).

We can recall the information on any defined data type using `show` command, e.g.

```
+ show HDexception;
datatype HDexception
con  Hd : HDexception
iter _HDexceptionit : HDexception -> 'a -> 'a
comp _HDexceptionit Hd = fn z => z
rec  _HDexceptionrec = _HDexceptionit
```

As an example of coinductive data type we define polymorphic stream (infinite lazy list). Coinductive data types are defined by a set of destructors (which encode elimination rules). Omitting `to` and a type expression in a destructor declaration is equivalent to a declaration of a destructor to the empty data type {}. A range of a destructor is a union of all types listed after the keyword `to`, hence if this list with preceding `to` is omitted, it is treated as the empty list, and the range is the empty union (i.e. the empty data type {}). A list of constructors (destructors) is optional. A datatype (codatatype) with no constructors (destructors) could be defined, which gives the possibility to define the minimal (maximal) type in the lattice of all types.

```
+ codatatype STREAM 'a = Shd to 'a & Stl to STREAM 'a;
codatatype STREAM 'a
des    Shd : (STREAM 'a) -> 'a
des    Stl : (STREAM 'a) -> STREAM 'a
coiter _STREAMci : ('b -> 'a) -> ('b -> 'b) -> 'b -> STREAM 'a
comp   Shd (_STREAMci z y x) = z x
comp   Stl (_STREAMci z y x) = _STREAMci z y (y x)
corec  _STREAMcr : ('b -> 'a) -> ('b -> (STREAM 'a) + 'b) -> 'b -> STREAM 'a
comp   Shd (_STREAMcr z y x) = z x
comp   Stl (_STREAMcr z y x) = when (y x) (fn w => w) (_STREAMcr z y)
+ val smap = fn f s => _STREAMci (fn x => f (Shd x)) Stl s;
val smap : ('a -> 'b) -> (STREAM 'a) -> STREAM 'b
```

`natstr` is a stream of increasing natural numbers, starting at some given number.

```
+ val natstr = fn n => _STREAMci (fn x => x) Suc n;
val natstr : NAT -> STREAM NAT
```

The function `sprefix` takes an element `h` and a stream `s` and puts `h` as the first element of the stream.

```
+ val sprefix = fn h s => _STREAMcr (fn x => h) (fn x => Inl s) s;
val sprefix : 'a -> (STREAM 'a) -> STREAM 'a
```

Let us experiment with `natstr`.

```
+ val ns = natstr O;
val ns : STREAM NAT
+ Shd ns;
O : NAT
+ Shd (Stl ns);
Suc O : NAT
+ val ns' = sprefix (Suc(Suc O)) ns;
val ns' : STREAM NAT
+ Shd ns';
Suc (Suc O) : NAT
+ Shd (Stl ns');
O : NAT
```

The next example is more involved. We define the function `run` which produces the list of entries of a binary labelled tree by breadth traversal[1]. It uses the data type `CONT 'a` which is positive, but not strictly positive, since it appears in a doubly negated position in the argument type of its constructor `CC`.

```
+ datatype CHAR = A|B|C|D|E|F|G|H|I|J|K|L|M;
datatype CHAR
con  A : CHAR
con  B : CHAR
con  C : CHAR
con  D : CHAR
con  E : CHAR
con  F : CHAR
con  G : CHAR
con  H : CHAR
con  I : CHAR
con  J : CHAR
con  K : CHAR
con  L : CHAR
con  M : CHAR
iter _CHARit : CHAR -> 'a -> 'a -> 'a -> 'a -> 'a -> 'a -> 'a -> 'a -> 'a ->
'a -> 'a -> 'a -> 'a -> 'a
comp _CHARit A = fn z y x w v u t s r q p o n => z
comp _CHARit B = fn z y x w v u t s r q p o n => y
comp _CHARit C = fn z y x w v u t s r q p o n => x
comp _CHARit D = fn z y x w v u t s r q p o n => w
comp _CHARit E = fn z y x w v u t s r q p o n => v
comp _CHARit F = fn z y x w v u t s r q p o n => u
comp _CHARit G = fn z y x w v u t s r q p o n => t
comp _CHARit H = fn z y x w v u t s r q p o n => s
comp _CHARit I = fn z y x w v u t s r q p o n => r
comp _CHARit J = fn z y x w v u t s r q p o n => q
comp _CHARit K = fn z y x w v u t s r q p o n => p
comp _CHARit L = fn z y x w v u t s r q p o n => o
comp _CHARit M = fn z y x w v u t s r q p o n => n
rec  _CHARrec = _CHARit
```

[1]It is a transformation of Martin Hofmann's program (in Lego notation). Message on the e-mail discussion list on types, February 1993.

```
+ datatype TREE 'a = Leaf from 'a | Node from 'a (TREE 'a) (TREE 'a);
datatype TREE 'a
con  Leaf : 'a -> TREE 'a
con  Node : 'a -> (TREE 'a) -> (TREE 'a) -> TREE 'a
iter _TREEit : (TREE 'a) -> ('a -> 'b) -> ('a -> 'b -> 'b -> 'b) -> 'b
comp _TREEit (Leaf z) = fn y x => y z
comp _TREEit (Node z y x) = fn w v => v z (_TREEit y w v) (_TREEit x w v)
rec  _TREErec : (TREE 'a) -> ('a -> 'b) -> ('a -> (TREE 'a) * 'b -> (TREE 'a) *
'b -> 'b) -> 'b
comp _TREErec (Leaf z) = fn y x => y z
comp _TREErec (Node z y x) = fn w v => v z (y , _TREErec y w v) (x , _TREErec
x w v)

+ datatype CONT 'a = DC | CC from ((CONT 'a) -> LIST 'a) -> LIST 'a;
datatype CONT 'a
con  DC : CONT 'a
con  CC : (((CONT 'a) -> LIST 'a) -> LIST 'a) -> CONT 'a
iter _CONTit : (CONT 'a) -> 'b -> ((('b -> LIST 'a) -> LIST 'a) -> 'b) -> 'b
comp _CONTit DC = fn z y => z
comp _CONTit (CC z) = fn y x => x (fn w => z (fn v => w (_CONTit v y x)))
rec  _CONTrec : (CONT 'a) -> 'b -> ((((CONT 'a) * 'b -> LIST 'a) -> LIST 'a) ->
 'b) -> 'b
comp _CONTrec DC = fn z y => z
comp _CONTrec (CC z) = fn y x => x (fn w => z (fn v => w (v , _CONTrec v y x)))

+ val o = fn g f x => g ( f x );
val o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b

+ val apply = fn k => _CONTit k (fn g => g DC)
=                               (fn h g => h (fn f => g ( CC f))
=                               );
val apply : (CONT 'a) -> ((CONT 'a) -> LIST 'a) -> LIST 'a

+ val breadth = fn t => _TREEit t
=                 (fn a now => CC (fn later => Cons a (apply now later)))
=                 (fn a brl brr now =>
=                    CC (fn later => Cons a (apply now (o later (o brl brr))))
=                 );
val breadth : (TREE 'a) -> (CONT 'a) -> CONT 'a

+ val extract = fn k => _CONTit k Nil (fn f => f (fn l => l));
val extract : (CONT 'a) -> LIST 'a

+ val run = fn t => extract (breadth t DC);
val run : (TREE 'a) -> LIST 'a

+ val t = Node A
=               (Node B
=                      (Node D (Leaf H) (Leaf I))
=                      (Node E (Leaf J) (Leaf K))
=               )
=               (Node C
=                      (Leaf F)
=                      (Node G (Leaf L) (Leaf M))
=               );
val t : TREE CHAR
```

```
+ run t;
Cons A (Cons B (Cons C (Cons D (Cons E (Cons F (Cons G (Cons H (Cons I (Cons J
(Cons K (Cons L (Cons M Nil))))))))))))) : LIST CHAR
```

The last example suggests the lines, along which IPL can be extended with linear types. Let us define two products.

```
+ datatype PROD1 'a 'b = Pair from 'a 'b;
datatype PROD1 'a 'b
con  Pair : 'a -> 'b -> PROD1 'a 'b
iter _PROD1it : (PROD1 'a 'b) -> ('a -> 'b -> 'c) -> 'c
comp _PROD1it (Pair z y) = fn x => x z y
rec  _PROD1rec = _PROD1it

+ codatatype PROD2 'a 'b = Fst to 'a & Snd to 'b;
codatatype PROD2 'a 'b
des    Fst : (PROD2 'a 'b) -> 'a
des    Snd : (PROD2 'a 'b) -> 'b
coiter _PROD2ci : ('c -> 'a) -> ('c -> 'b) -> 'c -> PROD2 'a 'b
comp   Fst (_PROD2ci z y x) = z x
comp   Snd (_PROD2ci z y x) = y x
corec  _PROD2cr = _PROD2ci
```

In IPL we can define `PROD1` in terms of `PROD2` and prove correctness of this definition.

```
+ val pair = fn x y => _PROD2ci (fn z => x) (fn z => y) ();
val pair : 'a -> 'b -> PROD2 'a 'b
+ val split = fn p z => z (Fst p) (Snd p);
val split : (PROD2 'a 'b) -> ('a -> 'b -> 'c) -> 'c
+ (fn x y => _PROD1it (Pair x y)) = (fn x y => split (pair x y));
True : BOOL
```

Similarly, `PROD2` can be defined in terms of `PROD1`.

```
+ val prod = fn f1 f2 x => Pair (f1 x) (f2 x);
val prod : ('a -> 'b) -> ('a -> 'c) -> 'a -> PROD1 'b 'c
+ val fst' = fn p => _PROD1it p (fn x y => x);
val fst' : (PROD1 'a 'b) -> 'a
+ val snd' = fn p => _PROD1it p (fn x y => y);
val snd' : (PROD1 'a 'b) -> 'b
+ (fn f g x => Fst (_PROD2ci f g x)) = (fn f g x => fst' (prod f g x));
True : BOOL
+ (fn f g x => Snd (_PROD2ci f g x)) = (fn f g x => snd' (prod f g x));
True : BOOL
```

In Girard's linear logic these products are distinct. `PROD1` would correspond to bilinear tensor product $\otimes$, and `PROD2` to linear product &. This may be explained in terms of resources. Suppose we have *one* entity of a resource `'c` and two functions `z : 'c -> 'a` and `y : 'c -> 'b` producing `'a` or `'b`, respectively, from this single resource. Then `_PROD2ci z y : 'c -> PROD2 'a 'b` consumes the resource `'c` and produces either `'a` or `'b`; we decide between them using `Fst` or `Snd`. On the other hand in the computation rule for `PROD1` both components of a pair are used.

In IPL resources are not limited, hence both products are equivalent — but this example gives some hints, how intuitionistic linear types can be added to IPL. It seems to me, that this should not be too difficult. The ideas of Girard's system LU[2], as explained by Wadler[3] would be useful.

[2]J.-Y. Girard. On the unity of logic. *Annals of Pure and Applied Logic*, 59(3):201–217, 1993.

[3]P. Wadler. A taste of linear logic. In A.M. Borzyszkowski and S. Sokołowski, editors, *Mathematical Foundations of Computer Science 1993*, pages 185–210. Springer-Verlag, Berlin, 1993. LNCS 711.