On the Implementation of IPL

Tomasz Wierzbicki
University of Wrocław
e-mail: `tomasz@ii.uni.wroc.pl`

In these notes I describe the first implementation of an experimental language invented by Zdzisław Spławski [2]. I wrote the IPL interpreter in Standard ML Core Language during a computer laboratory of a course on *Typed Lambda Calculi* using Edinburgh ML compiler v. 4.0. The first version of the program was presented at *Seminarium Warszawsko – Wrocławskie* in November, 1992. Since then some changes have been made (cotypes have been added etc.) and now a new version, proudly called “ver. 2.0”, is available.

In the following I describe technical details I had to establish (e.g. how the system generates names of eliminators etc.) They are inessential from the theoretical point of view and it is too early to define them permanently. Thus it is an example how the language *may* look rather than how it *will* look.

In a definition of the syntax a variant of the *BNF* has been used. Terminal symbols are printed in a `teletype` font. *Slanted* text states for non-terminal symbols. Meta-alternatives are listed between two vertical lines. [ ] denotes an option. { } repeats its contents at least once. A structure of the definitions corresponds to a precedence and associativity of operators. Comments, in general, describe only differences between SML and IPL.

$$\begin{array}{rcl} \textit{program} & = & [\ \{\ \textit{declaration}\ \}\ ] \\ \textit{declaration} & = & [\ \left|\begin{array}{l} \textit{value\_binding}\\ \textit{term}\\ \textit{datatype\_definition}\\ \textit{codatatype\_definition}\end{array}\right|\ ]\ ; \end{array}$$

An occurrence of a single term in a program is also a value binding: a value of this term is bound to the name `it` (and, of course, type-checked, evaluated (i.e. normalized) and then displayed (including functions.)) Thus a program is simply a sequence (may also be empty) of declarations. Note that a semicolon, unlike in SML, *must* occur at the end of any declaration.

$$
\begin{array}{rcl}
\mathit{atomic\_term} & = & \left| \begin{array}{l}
\mathit{value\_name} \\
\mathit{parameter} \\
\mathit{constructor} \\
\mathit{destructor} \\
\mathit{iterator} \\
\mathit{recursor} \\
\mathit{coiterator} \\
\mathit{corecursor} \\
\texttt{()} \\
\texttt{case0} \\
\texttt{case1} \\
\texttt{True} \\
\texttt{False} \\
\texttt{fst} \\
\texttt{snd} \\
\texttt{Inl} \\
\texttt{Inr} \\
\texttt{when} \\
\texttt{(}\ \mathit{term}\ \texttt{)} \\
\texttt{let}\ \{\ \mathit{value\_binding}\ \texttt{;}\ \}\ \texttt{in}\ \mathit{term}\ \texttt{end} \\
\texttt{fn}\ \{\ \mathit{parameter}\ \}\ \texttt{=>}\ \mathit{term} \\
\texttt{if}\ \mathit{term}\ \texttt{then}\ \mathit{term}\ \texttt{else}\ \mathit{term}
\end{array} \right| \\
\mathit{application} & = & [\ \mathit{application}\ ]\ \mathit{atomic\_term} \\
\mathit{pair} & = & [\ \mathit{pair}\ \texttt{,}\ ]\ \mathit{application} \\
\mathit{term} & = & [\ \mathit{term}\ \texttt{=}\ ]\ \mathit{pair}
\end{array}
$$

An application, pair constructor `,` and metapredicate `=` associate to the left. An application is the strongest, `=` — the weakest. The common rules of missing parentheses in lambda terms have been adopted. A body M of a lambda abstraction `fn` x `=>` M and an else branch of an if-then-else expression extend as far to the right as possible. `fn` $x_1 \ldots x_n$ `=>` M (inadmissible in SML) is equivalent to `fn` x_1 `=> ... => fn` x_n `=>` M. Note that in IPL fewer parentheses are needed than in SML, e.g.

```
fn x => x fn y => y
```

is valid and is equivalent to

```
fn x => x (fn y => y)
```

$$
\begin{array}{rcl}
\mathit{atomic\_type} & = & \left|\begin{array}{l}
\mathit{type\_variable}\\
\texttt{(}\ \mathit{type}\ \texttt{)}\\
\mathit{datatype\_constructor}\ [\ \{\ \mathit{type}\ \}\ ]\\
\mathit{codatatype\_constructor}\ [\ \{\ \mathit{type}\ \}\ ]\\
\{\}\\
\texttt{UNIT}\\
\texttt{BOOL}
\end{array}\right|\\[1ex]
\mathit{pair\_type} & = & [\ \mathit{pair\_type}\ \texttt{*}\ ]\ \mathit{atomic\_type}\\[1ex]
\mathit{union\_type} & = & [\ \mathit{union\_type}\ \texttt{+}\ ]\ \mathit{pair\_type}\\[1ex]
\mathit{type} & = & \mathit{union\_type}\ [\ \texttt{->}\ \mathit{type}\ ]
\end{array}
$$

A function type constructor `->` associates to the right and is weaker than a union type constructor `+` which is weaker than a pair type constructor `*`. Union an pair type constructors associate to the left. An order of a type concretization is determined by a number of type arguments, e.g. $T_2\ T_1\ T_0\ T_0$ means $T_2\ (\ T_1\ T_0\ )\ T_0$ if the type T_i has i arguments or $T_2\ (\ T_1\ T_0\ T_0\ )$ if T_0 has no arguments, T_1 — two, and T_2 — one. Type arguments are listed *after* a type constructor and there are no separators (like `(,,,)` in SML) between them. In responses, for readability, system uses more parentheses than required, but fewer than Edinburgh SML does.

$$
\mathit{value\_binding} \quad = \quad \texttt{val}\ \mathit{value\_name}\ \texttt{=}\ \mathit{term}
$$

Note that there is no function binding of the form `fun f x =` M

$$
\texttt{val f = fn x =>}\ M
$$

should be used instead of it. A defined value name (which occurs at the left side of a sign of equality) cannot occur in the term (i.e. general recursion is not allowed.)

$$
\begin{array}{rcl}
\mathit{datatype\_definition} & = & \texttt{datatype}\ \mathit{datatype\_constructor}\ [\ \{\ \mathit{type\_variable}\ \}\ ]\ \texttt{=}\\
 & & [\ \mathit{constructor\_list}\ ]\\[1ex]
\mathit{codatatype\_definition} & = & \texttt{codatatype}\ \mathit{codatatype\_constructor}\ [\ \{\ \mathit{type\_variable}\ \}\ ]\ \texttt{=}\\
 & & [\ \mathit{destructor\_list}\ ]\\[1ex]
\mathit{constructor\_list} & = & \mathit{constructor}\ [\ \texttt{from}\ \{\ \mathit{type}\ \}\ ]\ [\ \texttt{|}\ \mathit{constructor\_list}\ ]\\[1ex]
\mathit{destructor\_list} & = & \mathit{destructor}\ [\ \texttt{to}\ \{\ \mathit{type}\ \}\ ]\ [\ \texttt{\&}\ \mathit{destructor\_list}\ ]
\end{array}
$$

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. Omitting `to` and a type expression in a destructor declaration is equivalent to a declaration of a destructor to the absurd type {}. A range of a destructor is a union of all types listed after the keyword `to`, hence if this list is omitted with preceding `to`, it is treated as empty list, and the range is the empty union (i.e. the absurd type {}.) A constructor/destructor list is optional (a (co)datatype with no constructors/destructors could be defined.) It makes possibiblity to define the maximal/minimal type in a lattice of all types.

value_name = *lower_case alphanum*

parameter = *lower_case alphanum*

iterator = _ *datatype_constructor* `it`

recursor = _ *datatype_constructor* `rec`

coiterator = _ *codatatype_constructor* `ci`

corecursor = _ *codatatype_constructor* `cr`

constructor = *upper_case alphanum*

destructor = *upper_case alphanum*

type_variable = `'` *alphanum*

datatype_constructor = *letter alphanum*

codatatype_constructor = *letter alphanum*

Object constructors begin with capital, value names and formal parameters — with small letter, (co)datatype constructors can begin with both capital and small letter. The system derives a name of an eliminator from a corresponding type constructor adding an underscore at the beginning of the identifier and `it` (for iterators,) `rec` (for recursors,) `ci` (for coiterators) or `cr` (for corecursors) at its end. In responses the system names type variables adding a small letter to an apostrophe starting from `a` by `b`, `c` an so on (i.e. `'a 'b 'c ...` , like in SML,) and formal parameters using small letters from `z` by `y`, `x` and so on (SML does not display terms including lambda abstractions thus this is only an IPL feature.)

The IPL is a theoretical formalism rather than a programming language of a practical usage, hence it does not contain any side effects, syntactic sugar etc. Nevertheless in our IPL interpreter we had to include the following four 'earthly-minded' commands, that may occur in programs between declarations:

`use` *string* `;` — is similar to the SML function `use`. A string cannot contain `"`.

`show` $[\ \left|\begin{array}{l}\textit{datatype\_constructor}\\ \textit{codatatype\_constructor}\end{array}\right|\ ]$ `;` — lists all value names and type constructors stored in an environment. If a type constructor occurs after the keyword `show`, information about that type is listed.

`del` { *value_name* } `;` — removes values bound to listed names from an environment.

`exit ;` — quits an interpretation of a source file or shuts down an interactive session (usually used in the second case.)

Note that the commands listed above are not functions and do not return any value.

The remaining syntax definitions are as follows:

$$\begin{array}{rcl}
\textit{letter} & = & \left|\begin{array}{l}\textit{upper\_case}\\ \textit{lower\_case}\end{array}\right| \\[1em]
\textit{alphanum} & = & [\,\{\left|\begin{array}{c}\textit{letter}\\ \_\\ '\end{array}\right|\}\,] \\[1em]
\textit{upper\_case} & = & \text{one of } \texttt{ABCDEFGHIJKLMNOPQRSTUVWXYZ} \\
\textit{lower\_case} & = & \text{one of } \texttt{abcdefghijklmnopqrstuvwxyz} \\
\textit{digit} & = & \text{one of } \texttt{0123456789} \\
\textit{string} & = & \texttt{"}\quad \text{any sequence of ASCII char's not containing } \texttt{"}\quad \texttt{"} \\
\textit{comment} & = & \texttt{(*}\quad \text{any sequence of ASCII char's not containing } \texttt{*)}\quad \texttt{*)}
\end{array}$$

No keyword (including `True` , `BOOL` , `fst` etc.) may be used as an identifier. Comments cannot be nested. `(* This is (* a comment *)` is a valid comment. Lexical separators are: space, tab, newline and comment. Separators *may* occur between any two items and only there. They *must* occur between two alphanumeric items (e.g. keywords and identifiers) and only there.

The system prompts a user with `+` (equivalent to `-` in SML) at the beginning of a line. Continuation lines are preceded by `=` (like in SML.) System responses are not preceded by any sign.

A term M is in *normal form* iff it does not contain any redexes. The IPL has the strong normalization property, hence any term has a normal form. The normalization procedure consists of a sequence of contractions of redexes. There are eight kinds of redexes:

1. $($`fn` x `=>` $M_1)\ M_2$ (beta redex.) Its evaluation consists in substitution of M_2 for all free occurrences of x in M_1. De Bruijn's representation of lambda terms is used. Thus there are no name conflicts and alpha conversion is not necessary;
2. `fn` x `=>` $M\ x$ where there are no free occurrences of x in M (eta redex.) This redex contracts to M;
3. $I\ (C\ M_1\ \ldots\ M_k)$ where I is an iterator of some defined datatype T, C is a constructor of the datatype T an $M_1, \ldots, M_k$ are terms of apropriate types. Contraction of this redex consists in replacing it by the right hand side of the computation rule generated for the constructor C, where terms $M_1, \ldots, M_k$ are substituted for apropriate variables. Note that there is actually infinite number of kinds of redexes of that form, one for each constructor of each defined datatype;
4. the same as in 3 but for recursors;
5. $D\ (I\ M_1\ \ldots\ M_k)$ where D is a destructor of some defined codatatype T, I is a coiterator of the codatatype T, and $M_1, \ldots, M_k$ are terms of apropriate types. Contraction of this redex consists in replacing it by the right hand side of the computation rule generated for the destructor D, where terms $M_1, \ldots, M_k$ are substituted for apropriate variables. Note that there is actually infinite number of kinds of redexes of that form, one for each destructor of each defined codatatype;
6. the same as in 5 but for corecursors;
7. `fst` (M_1, M_2) — contracts to M_1;
8. `snd` (M_1, M_2) — contracts to M_2.

In accordance with the diamond and the strong normalization properties, the reduction strategy is inessential. In our IPL interpreter we consistently use lazy evaluation, e.g. we contract the leftmost outermost redex (of any kind) first, because in connection with the graph reduction it is the most efficient evaluation order (and is easy to program!)

A function, pair, union, absurd, unit and Boolean types have been predefined. Only the first of them is absolutely necessary. Its type constructor is an infix operator `->` and its object constructor is a lambda abstraction. There is no eliminator for this type but a lambda application (which has no denotation and plays a role of a selector for this type.) According to the strong normalization property we could predefine a metapredicate `=` for any type (including functions):

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

thus we had to predeclare:

```
type BOOL = True | False;
```

There are only syntactical differences between an if-then-else expression and an ordinary eliminator for the type `BOOL` (the iterator and recursor are equal since `BOOL` is not inductive.) We have a strange consequence of that fact — in Miranda, for example, a function `if` is lazily evaluated to the *Weak Head Normal Form (WHNF)* hence it behaves like an 'imperative' if-then-else statement: at most one branch is evaluated, but IPL normalizes 'strongly', thus if a logical condition is unresolvable (i.e. contains free variables) then both branches are evaluated. This is the price we have to pay for a well-defined equality. We have predefined also pair and union types:

```
datatype * 'a 'b = , from 'a 'b;
datatype + 'a 'b = Inl from 'a | Inr from 'b;
```

(certainly it cannot be written explicitly in IPL), where `*`, `+` and `,` are infix type and object constructors, respectively. To denote in IPL a pair $<<x, y>, z>$ one simply writes `x,y,z` (`,` associates to the left.) There are no eliminators predefined for this type but selectors `fst` and `snd`. One can define himself an eliminator:

```
val split = fn z y => y (fst z) (snd z);
```

but in normal form only selectors can occur. For the union type an ordinary eliminator named by keyword `when` has been predefined (it is iterator and recursor since union datatype is not inductive.) Pairs (in current version) are necessary to generate recursors, and unions to generate corecursors.

Remaining datatypes have been predefined as follows:

```
datatype {} = ;
datatype UNIT = ();
```

Their eliminators are named by keywords `case0` and `case1` respectively. According to this scheme one can also define

```
val case2 = fn b t f => if b then t else f;
```

To use the interpreter firstly install the program following instructions included in `README` file, then type `ipl` from the Unix shell to run it. Try `exit` command in IPL state to return to SML. The following values are defined in SML:

```
ipl_env   = - : environment ref
ipl_clear = fn: unit -> unit
ipl_run   = fn: string -> string -> environment -> unit
ipl       = fn: unit -> unit
```

An IPL environment (e.g. all declarations made during an IPL session) is stored in `ipl_env`. `ipl_clear` resets it to predefined environment. `ipl_run` executes a program. It gets a name of an input file, a name of an output file and an initial environment as parameters. `ipl` is a simpified version of `ipl_run` which uses a standard input and output, and `ipl_env` as its environment. If, for example, you prepared an IPL program in file `myprog.ipl`, you may type in SML state

```
- ipl_run "myprog.ipl" "myprog.lst" (!ipl_env);
```

obtaining a compilation (interpretation) listing in file `myfile.lst` rather than on console. The listing requires some editing, since it usually contains lines exceeding 80 columns (use a standard Unix `fmt` program with option `-s` to split long lines). You may also open an interactive session:

```
- ipl();
```

and type IPL command

```
+ use "myprog.ipl";
```

to get the listing on standard output.

References

[1] **Reade C.**, *Elements of Functional Programming,* Addison-Wesley, Reading Mass., 1989.

[2] **Spławski Z.**, *Proof-Theoretic Approach to Inductive Definitions in ML-like Programming Language versus Second-Order Lambda Calculus*, doctoral dissertation, University of Wrocław, 1993.

[3] **Tofte M.**, *Four Lectures on Standard ML,* Laboratory for Foundations of Computer Science, University of Edinburgh, 1989.