Web appendix of papers about Linear $\lambda$-calculus and Reversible Automatic Combinators

Alberto Ciaffaglione$^1$, Furio Honsell$^1$, Marina Lenisa$^1$, and Ivan Scagnetto$^1$

$^1$Università di Udine, Italy, 
\{alberto.ciaffaglione,furio.honsell,marina.lenisa,ivan.scagnetto\}@uniud.it

1 Introduction

This document illustrates in some details our implementation in the programming language Erlang [4, 5] of partial involutions, $\lambda$-terms and their types, along the lines of a series of papers, namely, [1, 2, 3].

A usable web interface for the main Erlang functions is available at the following URL:
http://158.110.146.197:31780/automata/

2 Implementing LApp in Erlang

In this section we will outline in full details our Erlang [4, 5] implementation of Abramsky’s LApp operator. See [6] for an alternate reading of this operator as the categorical trace on a subcategory of the category of Relations. We recall from [2] that we built a lexical analyzer and a syntactical parser for the language of partial involutions in order to allow the user to write rewriting rules in a user-friendly way, namely, as strings like the following one (for combinator B):

"rrrX<->lrX, l1X<->r1rX, r11X<->rr1X"

Once parsed, the previous string will yield an internal representation as a list of pairs: each pair is a directed (->$\lambda$) rewriting rule (hence, there are twice the rules for each clause in the original string, i.e., one for each rewriting direction). Thus, combinator B will be represented internally by the following list of pairs:

\[
\begin{align*}
\{\{r,\{r,\{r,\{var,"X"\}\}\}\}\}, \{l,\{\{var,"X"\}\}\}\}, \\
\{\{l,\{var,"X"\}\}\}, \{r,\{\{var,"X"\}\}\}\}, \\
\{\{l,\{var,"X"\}\}\}, \{r,\{\{var,"X"\}\}\}\}, \\
\{r,\{l,\{\{var,"X"\}\}\}\}, \{1,\{\{var,"X"\}\}\}\}, \\
\{\{r,\{l,\{\{var,"X"\}\}\}\}, \{l,\{\{var,"X"\}\}\}\}, \\
\{r,\{l,\{\{var,"X"\}\}\}\}, \{l,\{\{var,"X"\}\}\}\}, \\
\{r,\{l,\{\{var,"X"\}\}\}\}, \{r,\{l,\{\{var,"X"\}\}\}\}\}, \\
\{r,\{l,\{\{var,"X"\}\}\}\}, \{r,\{l,\{\{var,"X"\}\}\}\}\}\}
\end{align*}
\]
In this appendix all the functions we will introduce will operate on the internal representation (since it is more convenient for coding purposes); hence, we will not describe the lexical analyzer and the parser which are automatically generated from the grammar specification of the language of partial involutions, using leex and y ecc (i.e., the Erlang versions of the well known tools Lex and Yacc).

2.1 Auxiliary functions

First of all, we begin by introducing some basic auxiliary functions allowing us to deal smoothly with tasks such as eliminating duplicates from a list, extracting variables occurring in a given term/list, generating a fresh variable etc.

2.1.1 Removing duplicates from a list

The recursive definition is self-explanatory once we notice that member(H,T) is the library function returning true if and only if H is a member of T.

```erlang
dd(L) ->
    case L of
        [] -> [];    
        [H|T] ->
            Tail=dd(T),
            Exists=lists:member(H,T),
            if
            End =
            true -> [H]++Tail
            End.
    end.
```

The true clause corresponds to the else branch of the conditional command in common imperative languages.

2.1.2 Computing the list of variables occurring in terms and rewriting rule sets

Given a term T built starting from variables and constructors of the language of partial involutions (namely, ϵ, r, l, and p), the following function definition allows us to recursively compute the list of variables occurring into T:

```erlang
vars(T) ->
    case T of
        e -> [];    
        {var, X} -> [X];
        {1,U} -> vars(U);
        {r,U} -> vars(U);
        {p,U1,U2} ->
            V1==[] -> V2;
            V2==[] -> V1;
            true -> dd(V1++V2)
            end.
    end.
```
Notice the use of the \texttt{dd} function defined in Section 2.1.1, in order to avoid duplicates in the \texttt{p} constructor case.

We then use \texttt{vars} in order to specify the function \texttt{ruleListVars} which returns the list of variables occurring in the list of rewriting rules \texttt{L} passed as a parameter:

\begin{verbatim}
ruleListVars(L) ->
  case L of
    [] -> [] ;
    [R1,R2]|Tail -> dd(vars(R1)+vars(R2)+ruleListVars(Tail))
end.
\end{verbatim}

### 2.1.3 Generating fresh variables to avoid variables clashes

In order to avoid clashes with variables of different rewriting rule sets having the same name, we need a fresh renaming mechanism. The following function definition accepts a list of variables \texttt{L} as a parameter and returns a variable with prefix \texttt{P} (second parameter) not occurring in \texttt{L}:

\begin{verbatim}
fresh(L,P) ->
  case L of
    [] -> lists:flatten(io_lib:format(string:concat(P,"~p" ), [1]));
    [Head|Tail] -> Var=fresh(Tail,P),
                  Len=length(Head),
                  LenP=length(P),
                  if Len>=LenP -> Prefix=string:substr(Head, 1, LenP),
                              if Prefix=P -> {HeadId,.,}={string:
                                               to_integer(string:substr(Head, LenP+1, Len))},
                              if HeadId=error -> Var;
                              true -> {VarId,.,}=string:
                                               to_integer(string:substr(Var, LenP+1, length(Var)));
                              if VarId=HeadId -> Var;
                              true -> lists:flatten(io_lib:format(string:concat(P,"~p" ), [HeadId+1]))
                     end
                     end
                  end;
    true -> Var
  end
end.
\end{verbatim}

Automatically generated variables have names of the following shape: \texttt{Xn}, where \texttt{n} is an integer. Thus, they can be easily identified.

The \texttt{fresh} function is used in the definition of \texttt{separateVars} which returns a list of substitutions in order to separate variables occurring in \texttt{Vars1} from those occurring in \texttt{Vars2}, using the prefix \texttt{P} for new variables:
The attentive reader should notice that the new variable `NewVar` is chosen fresh w.r.t. both `Vars1` and `Vars2`. Moreover, we append both `Var` and `NewVar` to the second argument in the recursive calls; hence, the whole mechanism will work even after previous fresh renamings.

### 2.1.4 Substitutions

In this section we deal with various kinds of substitutions. We start with a function returning $T[Y/X]$, i.e., the result of substituting $Y$ for $X$ in $T$:

```haskell
subTerm(X,Y,T) ->
case T of
  e -> e;
  {var, V} -> if X == V -> Y;
    true -> {var, V}
  end;
  {l, U} -> {l, subTerm(X,Y,U)};
  {r, U} -> {r, subTerm(X,Y,U)};
  {p, P1, P2} -> {p, subTerm(X,Y,P1), subTerm(X,Y,P2)}
end.
```

Then, `subTerm` is used to implement two other types of multiple substitutions, namely, `subListTerm` where a list $L$ of substitutions $[T1/X1], \ldots, [Tn/Xn]$ is applied to a term $U$, yielding as a result $(\ldots(U[T1/X1])\ldots[Tn/Xn])$:

```haskell
subListTerm(L,U) ->
case L of
  [] -> U;
  [X,Y | Tail] -> subListTerm(Tail,subTerm(X,Y,U))
end.
```

and `subList` which is a function allowing one to substitute $Y$ for $X$ in the codomain of a list $L$ of substitutions:

```haskell
subList(X,Y,L) ->
case L of
  [] -> [];
  [Z,U | T] -> [{Z,subTerm(X,Y,U)} | subList(X,Y,T)]
end.
```
The last form of substitution we need is the one implemented by function `
subListRuleset` (built on top of the previously defined `
subListTerm`):

```plaintext
1 subListRuleset (Subst, Ruleset) ->
  case Ruleset of
    []   -> [];
    [R1,R2] Tail -> [subListTerm (Subst, R1),
                    subListTerm (Subst, R2)] ++ subListRuleset (Subst, Tail)
  end.
```

in this case we are applying a list of substitutions represented by the parameter `Subst` to all the terms occurring in the rewriting rules of the `Ruleset` parameter.

### 2.2 Robinson’s unification algorithm

The unification algorithm, originally conceived by Robinson, is essential in order to apply in sequence rewriting rules:

```plaintext
1 unify (X,Y,L) ->
  case X of
    e   -> case Y of
          e   -> {ok,L};
          {var,} -> unify (Y,X,L);
          _    -> {fail,[]}
      end;
    {var,V} -> Check=not(lists:member(V,vars(Y)));
               if Check -> {ok,dd(lists:append(subList(V,Y,L),[{V,Y}]))};
               true -> {fail,[]}
      end;
    {l,U}  -> case Y of
             {var,} -> unify (Y,X,L);
             {l,V}  -> unify (U,V,L);
             _     -> {fail,[]}
        end;
    {r,U}  -> case Y of
             {var,} -> unify (Y,X,L);
             {r,V}  -> unify (U,V,L);
             _     -> {fail,[]}
        end;
    {p,P1,P2} -> case Y of
                 {var,} -> unify (Y,X,L);
                 {p,Q1,Q2} -> {Flag1=R1}=unify (P1,Q1,L),
                               if
                               (Flag1==ok) -> {Flag2=R2}=unify (subListTerm(R1,P2),
                                                    subListTerm(R1,Q2),R1),
                               if
                               (Flag2==ok) -> { }
                 ok,R2};
               true -> {fail,[]}
               end;
               true -> {fail,[]}
               end;
    _    -> {fail,[]}
  end.
```
The intended meaning of \texttt{unify(X,Y,L)} is that the list \(L\) will be enriched by the substitutions needed to unify \(X\) and \(Y\). Hence, if we start with the empty list, we will obtain the m.g.u. of \(X\) and \(Y\). More precisely, the returned value is either a pair \{\texttt{ok,mgu}\} in case of success, or a pair \{\texttt{fail,[]}\} if \(X\) and \(Y\) are not unifiable.

### 2.3 Implementing \texttt{LApp}

As originally introduced by Abramsky, the definition of \texttt{LApp}(\(f,g\)) is given by

\[
\text{\texttt{extract}}(T, \texttt{Op1}, \texttt{Op2}) \rightarrow \\
\begin{cases}
\text{true} & \text{if } (\texttt{Op1}==\texttt{Op2}) \\
\text{false} & \text{otherwise}
\end{cases}
\]

Hence, we must begin programming a function named \texttt{extract}, being able to deduce l- and r-rewriting rules from \(L\), according to \texttt{Op1} and \texttt{Op2}:

```latex
\texttt{extract}(L,\texttt{Op1},\texttt{Op2}) \rightarrow \\
\begin{cases}
\text{\texttt{ok}} & \text{if } (\text{\texttt{unify}(R2,S1,[])}==\texttt{ok}) \\
\text{\texttt{fail}} & \text{otherwise}
\end{cases}
\]
```

Thus, if \(F\) represents a partial involution, then \texttt{extract}(\(F,r,l\)) will compute \(F_{rl}\).

Then, we have the \texttt{core} function \texttt{composeRuleList} which composes rule \(R1\rightarrow R2\) with all the rules in \(L\) (exploiting the unification and substitution functions defined in the previous sections):

```latex
\texttt{composeRuleList}(R1,R2,L) \rightarrow \\
\begin{cases}
\text{\texttt{ok}} & \text{if } (\text{\texttt{unify}(R2,S1,[])}==\texttt{ok}) \\
\text{\texttt{fail}} & \text{otherwise}
\end{cases}
\]
```
In order to avoid possible variable names clashes, the \texttt{alpha} function defined below replaces all variables in Ruleset1 which also occur in Ruleset2 with freshly generated ones:

\begin{verbatim}
alpha(Ruleset1, Ruleset2) ->
  Vars1=ruleListVars(Ruleset1),
  Vars2=ruleListVars(Ruleset2),
  FreshSubst=separateVars(Vars1, Vars2),
  subListRuleset(FreshSubst, Ruleset1).
\end{verbatim}

\texttt{alpha} is fruitfully used in the definition of \texttt{compose} which computes all possible chainings between rewriting rules of \texttt{L1} and \texttt{L2}:

\begin{verbatim}
compose(L1, L2) ->
  L1_Fresh=alpha(L1, L2),
  compose_fresh(L1_Fresh, L2).
\end{verbatim}

The function \texttt{star} will capitalize on the definition of \texttt{compose}, in order to implement the computation of \(H;(F;G)^*\):

\begin{verbatim}
star(H, F, G) ->
  S=compose(H, F),
  if S=[] -> H,
  true -> T=compose(S, G),
  if T=[] -> H,
  true -> H++star(T, F, G)
end.
\end{verbatim}

So far, the definition of our implementation of \texttt{LApp}, according to Abramsky’s specification, is straightforward:

\begin{verbatim}
lapp(F, G) ->
  FRR=extract(F, r, r),
  FRL=extract(F, r, l),
  FLL=extract(F, l, l),
  FLR=extract(F, l, r),
  FRL_G=compose(FRL, G),
  FRL_G_STAR=star(FRL_G, FLL, G),
  FRR++compose(FRL_G_STAR, FLR).
\end{verbatim}

In order to avoid an excessive nesting of \texttt{lapp} applications when dealing with complicate expressions, we also implemented the following functions:
Thus, we can delegate to `chainApp` the task of appropriately nesting the calls to `lapp`; indeed, the value computed by `chainApp([R1,R2,...,Rn])` is the rule set given by `lapp(...lapp(R1,R2)...),Rn).

## 2.4 Implementing replication

The last operator we need is replication: \( !f = \{(t,u),(t,v)\mid t \in T_\Sigma \land (u,v) \in f\} \). Its implementation in Erlang is straightforward:

```
1 bang(F) ->
2     Vars=ruleListVars(F),
3     X=fresh(Vars),
4     bang_rec(F,X).

5 bang_rec(F,X) ->
6     case F of
7         [] -> [ ];
8         [R] -> R;
9         [R | Tail] -> Rec=chain(Tail),
10        lapp(Rec,R)
11     end.
```

Notice how the fresh variable \( X \) plays the role of the generic term \( t \) in the original definition. Indeed, being a new variable, \( X \) can be unified with every possible term.

## 3 From \( \lambda \)-terms to combinators

In this section we will illustrate the details of the functions needed to transform plain \( \lambda \)-terms to expressions involving only combinators. As for the case of representing partial involutions, we built a lexer and parser for \( \lambda \)-terms and, again, we will not enter into the details of these programs. In the following table, we give a map from \( \lambda \)-terms “on the paper” to the corresponding input syntax, and finally to the internal representation:
<table>
<thead>
<tr>
<th>On the paper</th>
<th>Input syntax</th>
<th>Internal representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>X</td>
<td>{var, &quot;X&quot;}</td>
</tr>
<tr>
<td>$C$</td>
<td>C</td>
<td>{comb, C}</td>
</tr>
<tr>
<td>$MN$</td>
<td>$M \otimes N$</td>
<td>{lapp, M, N}</td>
</tr>
<tr>
<td>$\lambda x.M$</td>
<td>$1 \cdot X.M$</td>
<td>{abs, {var, &quot;X&quot;}, M}</td>
</tr>
<tr>
<td>$\lambda !x.M$</td>
<td>$1 \cdot !X.M$</td>
<td>{abs_b, {var, &quot;X&quot;}, M}</td>
</tr>
<tr>
<td>$!M$</td>
<td>!M</td>
<td>{bang, M}</td>
</tr>
</tbody>
</table>

### 3.1 Auxiliary functions

#### 3.1.1 Renaming automatic variables

In order to do not interfere with variable names assigned by users, we reserve identifiers of the shape $X_i$ (where $i$ is a natural number $\geq 1$) for automatic variables (i.e., variables generated in a programmatic way). Thus, we can easily rename such variables by increasing the index $i$. This is precisely the purpose of the function `shift` which increases the index of the first argument by the amount specified by the second argument:

```plaintext
shift(Var, 1) ->
Len=length(Var),
if Len>=2 ->
  Prefix=string:substr(Var, 1, 2),
  if Prefix=="X" ->
    {VarId,}={var, Fv},
    if VarId==error -> Var;
    true -> lists:flatten(io_lib:format("X\^p", [VarId+1]))
  end;
  true -> Var
end;
true -> Var
end.
```

#### 3.1.2 Function computing the substitution needed to rename (with $Fv$, shift($Fv, 1$), shift($Fv, 2$), ...) all variables occurring in a list of variables

The function `compute_renaming` uses the previously defined function `shift` (see Section 3.1.1), in order to transform a list of variables $[X, Y, Z, ...]$ into $[Fv, shift(Fv, 1), shift(Fv, 2), ...]$. The function is defined as follows:

```plaintext
compute_renaming(Vars, Fv) ->
  case Vars of
    [ ] -> [ ];
    [V | Tail] -> [{V, {var, Fv}} | compute_renaming(Tail, shift(Fv, 1))]
  end.
```
3.1.3 Function renaming (with $Fv$, $\text{shift}(Fv,1)$, $\text{shift}(Fv,2)$, ...) all variables occurring in the ruleset $R$

Using \texttt{compute_renaming} we can proceed further with the renaming of all the variables occurring in a ruleset by $Fv$, $\text{shift}(Fv,1)$, $\text{shift}(Fv,2)$, ...

\begin{verbatim}
   polish_rules(R, Fv) ->
   case R of
      [] -> [];
      [R1,R2] | Tail  -> Vars=ruleListVars([[R1,R2]]),
                  Subst=compute_renaming(Vars,Fv),
                  [subListTerm(Subst,R1),subListTerm(Subst,
                  R2)] | polish_rules(Tail,Fv)
   end.
\end{verbatim}

3.1.4 Polishing a ruleset $R$

The following function generates a new fresh variable and it uses the latter as a basis for renaming all variables of the ruleset $R$, removing also possible duplicate clauses.

\begin{verbatim}
   polish(R) ->
   Vars=ruleListVars(R),
   Fv=fresh(Vars),
   dd(polish_rules(R,Fv)).
\end{verbatim}

3.1.5 Deciding if two rulesets are equivalent

So far, it is rather straightforward to check if two given rulesets $R1$ and $R2$ are equivalent:

1. choose a fresh variable w.r.t. both variables in $R1$, $R2$;
2. polish both rulesets, removing possible duplicates;
3. test if the results are equal.

\begin{verbatim}
   equiv(R1,R2) ->
   Vars1=ruleListVars(R1),
   Vars2=ruleListVars(R2),
   Fv=fresh(Vars1++Vars2),
   P1=dd(polish_rules(R1,Fv)),
   P2=dd(polish_rules(R2,Fv)),
   lists:sort(P1) == lists:sort(P2).
\end{verbatim}

3.1.6 Computing the list of the free variables of a $\lambda$-term

Our last auxiliary function computes the list of the free variables occurring in a $\lambda$-term.
3.2 Abstraction operator

The implementation of the Abstraction Operator, extended to the full λ-calculus, allows the user to “translate” a λ-term into an expression built using only combinators. This is precisely the purpose of `abstract`:

```haskell
abstract (Lambda) ->
case Lambda of
    { var , X } -> [X];
    { comb , } -> [];
    { lapp , M , N } -> dd (varsLambda (M) ++ varsLambda (N)) ;
    { abs , { var , X } , M } -> lists : delete (X , varsLambda (M));
    { bang , M } -> varsLambda (M)
end .
```
The previous code implements the abstraction algorithm $\lambda^f$ for the general case of the $\lambda$-calculus (see, e.g., [1]). On the other hand, if one is interested in the Milner, EAL or LAL cases, the abstraction algorithm changes, applying other ad-hoc transformation rules.

## 4 Combinators as partial involutions

### 4.1 Standard Combinators

The purpose of the following function is to return a list whose members are the internal representation as partial involutions of the standard combinators of Combinatory Logic.

```erlang
export_combinators() ->
  % Combinators as sets of rewriting rules.
  {string, 1} = "1X<->\pi X",
  {string, 2} = "rrX<->lrX, l1X<->r1LX, r1LX<->r1LX",
  {string, 3} = "llX<->r1LX, lrlX<->r1LX, lrlrX<->rrLX",
  {string, 4} = "1<e,X<->\pi X",
  {string, 5} = "rrX<->lrX, l1X<->r1LX, r1LX<->r1LX, rrlX<->rrLX, r1LX<->l1LX"
```
4.2 Decoding combinators expressions into partial involutions

The result yielded by export_combinators can be fed to decode in order to transform a combinatory logic term into linear and bang applications (yielding a partial involution as the final result).

```haskell
decode (TComb, Combinators) ->
case TComb of
  {comb, Comb} -> Check = lists:keyfind(Comb, 1, Combinators),
                 if Check = false -> [];
                 true -> element(2, Check)
         end;
  {lapp, M, N} -> lapp (decode(M, Combinators), decode(N, Combinators));
  {bang, M} -> bang (decode(M, Combinators))
end.
```

5 Testing and pretty printing

In this section we describe some auxiliary functions whose purpose it to ease the automation of tests (i.e., the equivalence checks between partial involutions), and to pretty print the results to the screen.
5.1 Pretty printing terms

```haskell
pretty_print_term(T) ->
case T of
e  -> io:format("e");
{var, V}  -> io:format(V);
{T1}  -> io:format("l("), pretty_print_term(T1), io:format(")");
r, T1  -> io:format("r("), pretty_print_term(T1), io:format(")");
p, T1, T2  -> io:format("<"), pretty_print_term(T1), io:format(",") , pretty_print_term(T2), io:format(">")
end.
```

5.2 Managing and pretty printing lists of rewriting rules

The following function erases all occurrences of rules \{A, B\} and \{B, A\} from L:

```haskell
erase_dup(A,B,L) ->
case L of
[]  -> [];
[A,B]|L1  -> erase_dup(A,B,L1);
[B,A]|L1  -> erase_dup(A,B,L1);
[C,D]|L1  -> [{C,D}]++erase_dup(A,B,L1)
end.
```

Sometimes we need to check for and erase duplicates of rule \{A, B\} in L, in order to avoid redoing the same computations for the terms A and B:

```haskell
check_dup(R) ->
case R of
[]  -> [];
[A,B]|T  -> T1=erase_dup(A,B,T),
       [{A,B}]++check_dup(T1)
end.
```

The following function adds inverted rewriting rules to a list R, in order to provide a partial involution as a result.

```haskell
dup(R) ->
case R of
[]  -> [];
[A,B]|T  -> [{A,B},{B,A}]++dup(T)
end.
```

In order to pretty print a partial involution, we can use the following function:

```haskell
pretty_print_rules(L) ->
case L of
[]  -> io:format("−−−−−− end −−−−−−n");
[R1,R2]|T  -> pretty_print_term(R1), io:format(" "->""), pretty_print_term(R2), io:format("n"), pretty_print_rules(T)
end.
```
5.3 Pretty printing lambda terms

```haskell
pretty_print_lambda (T) ->
case T of
  { var , X } -> io : format (X);
  { comb , C } -> io : format (C);
  { lapp , M , N } -> io : format ("( " ) , pretty_print_lambda (M) ,
                   pretty_print_lambda (N) , io : format (" ) ");
  { abs , { var , X } , M } -> io : format ("1 © " ) , io : format (X) , io : format (" . " ) , pretty_print_lambda (M);
  { abs_b , { var , X } , M } -> io : format ("1 © ! " ) , io : format (X) , io :
                   format (" " ) , pretty_print_lambda (M);
  { bang , M } -> io : format ("! ( " ) , pretty_print_lambda (M) , io : format (" ) ");
end .
```

5.4 Checking equivalence of partial involutions

The function `test` in the following returns `true` if and only if `Lambda_string1` and `Lambda_string2` are equivalent λ²-expressions.

```haskell
test ( Lambda_string1 , Lambda_string2 , Combinators ) ->
  { ok , Tokens , 1 } = lambda_lexer : string ( Lambda_string1 ) ,
  { ok , LambdaT1 } = lambda_parser : parse ( Tokens , LambdaT1 ) ,
  Comb_LambdaT1 = abstract ( LambdaT1 ) ,
  io : format ("− − − " ) , pretty_print_lambda ( Comb_LambdaT1 ) , io : format ("−−− " ) ,
  Test1 = decode ( Comb_LambdaT1 , Combinators ) ,
  pretty_print_rules ( Test1 ) ,
  { ok , Tokens , 1 } = lambda_lexer : string ( Lambda_string2 ) ,
  { ok , LambdaT2 } = lambda_parser : parse ( Tokens , LambdaT2 ) ,
  Comb_LambdaT2 = abstract ( LambdaT2 ) ,
  io : format ("− − − " ) , pretty_print_lambda ( Comb_LambdaT2 ) , io : format ("−−− " ) ,
  Test2 = decode ( Comb_LambdaT2 , Combinators ) ,
  pretty_print_rules ( Test2 ) ,
  equiv ( Test1 , Test2 ) .
```

5.5 Converting and pretty printing λ²-terms

The following function prints to the screen the combinators expression and the rewriting rules corresponding to the λ²-expression `LambdaString`.

```haskell
show ( LambdaString ) ->
  Combinators = export_combinators () ,
  { ok , Tokens , 1 } = lambda_lexer : string ( LambdaString ) ,
  { ok , T } = lambda_parser : parse ( Tokens ) ,
  Comb_T = abstract ( T ) ,
  pretty_print_lambda ( Comb_T ) , io : format (" " ) ,
```
6 Running example

In order to give an idea of how to use all the machinery so far introduced, we will consider an example session at the Erlang console.

Let us see how to prove that equation $\lambda^{xyz}.C(C(BBx)y)z = \lambda^{xyz}.Cz(yz)$ from Theorem 2 of [2] holds:

```erlang
1> Combinators=abramsky:export_combinators().
2> pretty_print_rules(polish(decode(Comb_T,Combinators))).
```

Pretty print rules (polish (decode (Comb_T, Combinators))).
Hence, the equation holds; indeed, the value returned by the function call 
abramsky:test(LambdaT1_string,LambdaT2_string,Combinators) at line 63
is true (line 84).

Notice again (see Section 2) that the rewriting rules of partial involutions 
\( A \leftrightarrow B \) are, for programming convenience reasons, internally represented by 
two expressions, namely, \( A \rightarrow B \) and \( B \rightarrow A \).

7 Types

As for the case of representing partial involutions and \( \lambda \)-terms, we built a lexer 
and parser for types and, again, we will not enter into the details of these 
programs. In the following table, we give a map from types “on the paper” to 
the corresponding input syntax, and finally to the internal representation:
7.1 Auxiliary Functions

7.1.1 From input syntax to internal representation

The following function prints the string representation of a type together with its internal representation for testing purposes.

```haskell
show_type :: TypeString ->
             (ok, Tokens, 1) = type Lexer: string (TypeString),
             {ok, T} = type Parser: parse (Tokens),
             io: format("p: n" ++ [TypeString, T]).
```

Instead, the following one is called within other functions as the first step to apply some algorithm to a type. It carries out lexing, parsing and it returns the type corresponding to the string representation passed as argument.

```haskell
string2type :: TypeString ->
             {ok, Tokens, 1} = type Lexer: string (TypeString),
             {ok, T} = type Parser: parse (Tokens),
             T.
```

7.1.2 Variables and variables occurrences

We can use the following function to obtain the list of free variables in a type:

```haskell
varsType :: Type ->
          case Type of
            omega -> [ ];
            {var, X} -> [X];
            {map, T1, T2} -> dd (varsType T1 ++ varsType T2);
            {cap, T1, T2} -> dd (varsType T1 ++ varsType T2);
            {bang, I, T} -> dd (vars I ++ varsType T)
          end.
```

If we are interested in the number of occurrences of a variable in a type, we can use the following function:

```haskell
varTypeCount :: Var, Type ->
               case Type of
                 {var, X} -> if
                       X == Var -> 1;
                       true -> 0
                 end;
                 {map, T1, T2} -> varTypeCount (Var, T1) + varTypeCount (Var, T2);
                 {cap, T1, T2} -> varTypeCount (Var, T1) + varTypeCount (Var, T2);
                 {bang, I, T} -> varTypeCount (Var, T)
               end.
```
7.2 From types to partial involutions

The following function accepts a type variable \( \text{Var} \) and a type representation \( \text{Type} \) as parameters, and it tries to find the rules based on \( \text{Var} \) of the p.i. corresponding to \( \text{Type} \).

\[
\text{findPI}(\text{Var}, \text{Type}) \rightarrow \\
\quad \text{PrefixCheck} = \text{str}(\text{Var}, "\Omega") , \\
\quad \text{case} \quad \text{PrefixCheck} = 1 \rightarrow [ ] ; \\
\quad \text{true} \rightarrow \text{var_type2PI}(\text{Var}, \text{Type}) , \\
\quad \text{if} \\
\quad \quad P = [ ] \rightarrow [ ] ; \\
\quad \quad \text{true} \rightarrow \text{pairwise}(P) \\
\quad \text{end} \\
\text{end} .
\]

The final rules are generated by computing all the possible pairs between terms by means of the following function:

\[
\text{pairwise}(L) \rightarrow \\
\quad \text{case} \quad L = [ ] \rightarrow [ ] ; \\
\quad \quad \text{case} \quad T = [ ] \rightarrow [ ] ; \\
\quad \quad \quad \quad \text{if} \\
\quad \quad \quad \quad \quad \text{PrefixCheck} = 1 \rightarrow [ ] ; \\
\quad \quad \quad \quad \quad \text{true} \rightarrow \text{pairwise}(\text{PrefixCheck}) \\
\quad \quad \quad \quad \quad \text{end} \\
\quad \quad \text{case} \quad T = [ \text{H1} | \text{T1}] \rightarrow \\
\quad \quad \quad \quad \quad \text{dd}([\text{H, H1}], [\text{H1, H}])++\text{pairwise}(\text{H1} | \text{T1})++\text{pairwise}(\text{H} | \text{T1}) \\
\quad \quad \text{end} \\
\quad \text{end} .
\]

However, the real work is carried out by \( \text{var_type2PI()} \) which actually computes the partial involution terms corresponding to the variable occurrences of \( \text{Var} \) in type \( \text{Type} \):

\[
\text{var_type2PI}(\text{Var}, \text{Type}) \rightarrow \\
\quad \text{case} \quad \text{Type} = \omega \rightarrow [ ] ; \\
\quad \quad \text{case} \quad \text{Type} = \{ \text{var}, X \} \rightarrow \\
\quad \quad \quad \quad \text{X} = \text{Var} \rightarrow [\{ \text{var}, \text{Var} \}] ; \\
\quad \quad \quad \quad \text{true} \rightarrow [ ] ; \\
\quad \quad \text{end} ; \\
\quad \quad \{ \text{map}, \text{T1}, \text{T2} \} \rightarrow \\
\quad \quad \quad \quad \text{CheckT1} = \text{lists}:\text{member}(\text{Var}, \text{varType}(\text{T1})) , \\
\quad \quad \quad \quad \text{CheckT2} = \text{lists}:\text{member}(\text{Var}, \text{varType}(\text{T2})) ; \\
\quad \quad \quad \quad \text{if} \\
\quad \quad \quad \quad \quad \text{CheckT1 and CheckT2} \rightarrow \text{var_type2PI}(\text{Var}, \text{T1}) , \\
\quad \quad \quad \quad \quad \text{P1=var_type2PI}(\text{Var}, \text{T1}) , \\
\quad \quad \quad \quad \quad \quad \text{distribute}_{lr}(l, \text{P1})++\text{distribute}_{lr}(r, \text{P2}) ; \\
\quad \quad \quad \quad \quad \text{CheckT1 and not(\text{CheckT2})} \rightarrow \text{var_type2PI}(\text{Var}, \text{T1}) , \\
\quad \quad \quad \quad \quad \quad \text{distribute}_{lr}(l, \text{P}) ; \\
\quad \quad \quad \quad \quad \text{CheckT2 and not(\text{CheckT1})} \rightarrow \text{var_type2PI}(\text{Var}, \text{T2}) , \\
\quad \quad \quad \quad \quad \quad \text{distribute}_{rl}(r, \text{P}) ; \\
\quad \quad \quad \quad \quad \text{true} \rightarrow [ ] \\
\quad \quad \quad \quad \quad \text{end} ; \\
\quad \quad \quad \quad \text{end} ; \\
\quad \quad \quad \quad \{ \text{cap}, \text{T1}, \text{T2} \} \rightarrow \\
\quad \quad \quad \quad \quad \text{Prefix} = \text{find_common_prefix}(\text{T1}, \text{T2}) ; \\
\quad \quad \quad \quad \quad \text{true} \rightarrow \text{checkT1 = lists:member}(\text{Var}, \text{varType}(\text{T1})) ; \\
\quad \quad \quad \quad \quad \text{checkT2 = lists:member}(\text{Var}, \text{varType}(\text{T2})) ; \\
\quad \quad \quad \quad \quad \text{if} \\
\quad \quad \quad \quad \quad \text{CheckT1 and CheckT2} \rightarrow \text{P1=var_type2PI}(\text{Var}, \text{T1}) , \\
\quad \quad \quad \quad \quad \text{P2=var_type2PI}(\text{Var}, \text{T2}) , \\
\quad \quad \quad \quad \quad \quad \text{compose}_{index}(l, \text{P1})++\text{compose}_{index}(r, \text{P2}) ; \\
\quad \quad \quad \quad \quad \text{CheckT1 and not(\text{CheckT2})} \rightarrow \text{P1=var_type2PI}(\text{Var}, \text{T1}) , \\
\quad \quad \quad \quad \quad \quad \text{compose}_{index}(l, \text{P}) ; \\
\quad \quad \quad \quad \quad \text{CheckT2 and not(\text{CheckT1})} \rightarrow \text{P2=var_type2PI}(\text{Var}, \text{T2}) , \\
\quad \quad \quad \quad \quad \quad \text{compose}_{index}(r, \text{P}) ; \\
\quad \quad \quad \quad \quad \text{true} \rightarrow [ ] \\
\quad \quad \quad \quad \quad \text{end} ; \\
\quad \quad \quad \quad \quad \text{end} ; \\
\quad \quad \quad \quad \{ \text{bang}, I, T \} \rightarrow \\
\quad \quad \quad \quad \quad \text{CheckT = lists:member}(\text{Var}, \text{varType}(\text{T})) ; \\
\quad \quad \quad \quad \quad \text{end} ; \\
\quad \text{end} ;
\]

19
The distribution of the constructors \( l \) and \( r \) in a list of partial involution terms, is carried out by `distribute_lr()`: 

```plaintext
distribute_lr(Constr, P) →
case P of
  [] → [];
  [A | Tail] → distribute_lr(Constr, A) ++ distribute_lr(Constr, Tail);
  A → [{Constr, A}]
end.
```

Similar operations on indexes are carried out by `add_index()` (where we pay attention to group together pair of indexes in front of \( l \)- and \( r \)- constructors): 

```plaintext
add_index(Index, P) →
case P of
  [] → [];
  [A | Tail] → case A of
    {p, I} → [{p, {Index, I}, {l, B}}] ++
               add_index(Index, Tail);
    {p, I} → [{p, {Index, I}, {r, B}}] ++
               add_index(Index, Tail);
  A → [{p, Index, A}] ++ add_index(Index, Tail)
end.
```

and by `compose_index()`:

```plaintext
compose_index(Constr, P) →
case P of
  [] → [];
  [p, Index, A] | Tail → [{p, {Constr, Index}, A}] ++
                         compose_index(Constr, Tail);
  A | Tail → compose_index(Constr, Tail)
end.
```

When processing a type, we need a function capable of extracting the common prefix of a series of types’ conjunctions:

```plaintext
find_common_prefix(T1, T2) →
case T1 of
  {bang, {p, I1, I2}, U1} → case T2 of
    {bang, {p, J1, J2}, U2} → if I1 :== J1 →
                           {cap, {bang, I2, U1}, {bang, J2, U2}};
                           true
                           -> {none, none}
        -> {cap, V1, V2} → {P, Res}=
find_common_prefix(V1, V2),
```
The function `processVariables()` exhaustively applies `findPI()` to each variable of type `Type` in the list `Vars`:

```plaintext
processVariables(Vars, Type) ->
  case Vars of
    [] -> [];
    [V|L] -> findPI(V, Type)++processVariables(L, Type)
  end.
```

Finally, we have the front-end function the user has to call, in order to start the whole process, calculating and pretty printing the partial involution corresponding to a given type (passed as a string):

```plaintext
show_partial_involution(TypeString) ->
  Type=string2type(TypeString),
  Vars=varsType(Type),
  Rules=processVariables(Vars, Type),
  pretty_print_rules(Rules).
```

### 7.3 Other auxiliary functions

#### 7.3.1 Substitutions of types in types

In order to substitute Y for X in T (yielding T[Y/X]), we can use the following function:
The following function applies a list of substitutions to type U (using the previously defined function subType()):

subListType(L, U) →
  case L of
    [] → U;
    [X,Y | Tail] → subListType(Tail, subType(X,Y,U))
  end.

As an alternative, if we need to substitute Y for X only in the codomain of a list L of substitutions for types, we have the function subList2():

subList2(X,Y,L) →
  case L of
    [] → [];  
    [Z,U | T] → [{Z, subType(X,Y,U)} | subList2(X,Y,T)]
  end.

### 7.3.2 Type variables

The following function returns a list of substitutions, in order to separate type variables occurring in Vars1 from those occurring in Vars2, distinguishing between X_i and \( \Omega \_i \) kinds of fresh variables.

separateTypeVars(Vars1, Vars2) →
  case Vars1 of
    [] → [];
    [Var|Tail] → Check=lists: member(Var, Vars2),
      if
        Check → Len=length(Var),
            LenOmega=length("\_Omega"),
            if
              Len<LenOmega → NewVar=fresh(Vars1++Vars2,"X")
                , true → Prefix=string: substr(Var, 1, LenOmega),
                if
                  Prefix="\_Omega" → NewVar=fresh(Vars1++Vars2,"\_Omega")
                    , true → NewVar=fresh(Vars1++Vars2,"X")
                    , end
              true → NewVar=fresh(Vars1++Vars2,"\_Omega")
                , true → NewVar=fresh(Vars1++Vars2,"X")
                , end
            if
      if
The following function returns a list of the \texttt{\_Oomega} kind of variables in the type \texttt{Type}:

\begin{verbatim}
omegaVarsType (Type) ->
case Type of
  omega -> [ ];
  {var, X} -> PrefixCheck=string: str (X, "\_Omega"),
  if PrefixCheck==1 -> [X];
  true -> []
end;
{map, T1, T2} -> dd(omegaVarsType(T1) ++ omegaVarsType(T2));
{cap, T1, T2} -> dd(omegaVarsType(T1) ++ omegaVarsType(T2));
{bang, _, T} -> omegaVarsType(T)
end.
\end{verbatim}

### 7.3.3 Pretty printing types

In order to have a user-friendly and human readable representation of types, we can use the following function to pretty print the internal representation:

\begin{verbatim}
pretty_print_type(T) ->
case T of
  omega -> io:format("omega");
  {var, V} -> io:format(V);
  {map, T1, T2} -> io:format("("), pretty_print_type(T1), io:format(" - ")
  /\ " ), pretty_print_type(T2), io:format("\)"));
  {cap, T1, T2} -> io:format("("), pretty_print_type(T1), io:format("/
  /\ " ), pretty_print_type(T2), io:format("\)"));
  {bang, I, T1} -> io:format("(!") , pretty_print_term(I) , io:format(""
  /\ " ), pretty_print_type(T1), io:format("\))")
end.
\end{verbatim}

### 7.4 Unification of types

The unification of types mainly follows the classic Robinson’s unification algorithm, but there are some subtle changes due to the semantics of intersection types. Indeed, the latter may “spawn” multiple copies of the involved types, in order to allow the unification process.

The following function tries to unify types \texttt{X} and \texttt{Y}, putting the needed substitutions of type variables in the list \texttt{L1}, and the needed substitutions of term variables in the list \texttt{L2}. The list \texttt{OmegaVarList} contains all the variables which must be taken distinct from eventually new generated variables, during the unification process.
unifyTypes(X, Y, OmegaVarList, L1, L2) →
  case X of
    omega → case Y of
      omega → (ok, L1, L2, X),
        (var, Z) → (ok, dd([lists: append(subList2(Z, omega, L1), [Z, omega]])), L2, X),
        → (fail, [], []), X)
    end;
  end;
  (var, V) → case Y of
    (var, V) → (ok, L1, L2, X),
    (var, Z) → PrefixCheck1 = string: str(V, "Omega"),
      PrefixCheck2 = string: str(Z, "Omega"),
      if (PrefixCheck1 == ok) and (PrefixCheck2 == ok) → unifyTypes
      (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])]), L2, Y}
    end;
    T → PrefixCheck1 = string: str(V, ",Omega"),
      PrefixCheck2 = string: str(T, ",Omega"),
      if (PrefixCheck1 == ok) → unifyTypes
      (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, T, L1), [V, T])]), L2, T};
    end;
  end;
end;

if (Flag1 == ok) → {Flag1 == ok};
  if (Flag2 == ok) → unifyTypes
    (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
  end;
end;

if (Flag1 == ok) → {Flag1 == ok};
  if (Flag2 == ok) → unifyTypes
    (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
  end;
end;

(i if
  if unifyTypes(X, Y, OmegaVarList, L1, L2);
    {map, U1, U2} → {Flag1, R1, TR1, NewX} = unifyTypes(T1, U1, OmegaVarList, L1, L2),
    if (Flag1 == ok) → {Flag1 == ok};
      if (Flag2 == ok) → unifyTypes
        (Y, X, OmegaVarList, L1, L2);
        true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
      end;
    end;
end;

i if
  if unifyTypes(X, Y, OmegaVarList, L1, L2);
    {map, U1, U2} → {Flag1, R1, TR1, NewX} = unifyTypes(T1, U1, OmegaVarList, L1, L2),
    if (Flag1 == ok) → {Flag1 == ok};
      if (Flag2 == ok) → unifyTypes
        (Y, X, OmegaVarList, L1, L2);
        true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
      end;
end;

% Trying to spawn more \ if unification fails...
  {bang, [], J} → Vars = dd([omegaTypes(X)++omegaTypes(Y)]+++OmegaVarList),
    Omega = fresh([Vars, "$Omega"],
      {Flag1, L3, L4, NewType} = unifyTypes(X, {map, Y, [var, Omega]}, L1, L2),
      if (Flag1 == ok) → {Flag1 == ok};
        Vars = dd([omegaTypes(X)++omegaTypes(Y)]+++OmegaVarList),
        true → unifyTypes(X, {map, [var, Omega]}, Y, Vars++,
          [Omega], L1, L2)
      end;
  end;

% Otherwise, fail...
  → (fail, [], []).
end;

if (Flag1 == ok) → {Flag1 == ok};
  if (Flag2 == ok) → unifyTypes
    (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
  end;
end;

if (Flag1 == ok) → {Flag1 == ok};
  if (Flag2 == ok) → unifyTypes
    (Y, X, OmegaVarList, L1, L2);
    true → {ok, dd([lists: append(subList2(V, Y, L1), [V, Y])])}, L2, Y
  end;
end;

24
% Trying to spawn more \ if unification fails...
{cap, ..} -> Vars++omegaVarType(X)++omegaVarType(Y)++
OmegaVarList), Omega=fresh(Vars,"\Omega"),
{Flag, L3, L4, NewType} = unifyTypes({cap, X, {var, Omega}
}, Y, Vars++[Omega], L1, L2),
if Flag==ok -> {Flag, L3, L4, NewType};
true -> unifyTypes({cap, {var, Omega}, X}, Y, Vars++[Omega], L1, L2)
end;
% Otherwise, fail...
{fail, [], [], X}
end.

Unification of indexes, in the \texttt{bang} case, is dealt with in the following function:

```plaintext
unify_index(X, Y, L) ->
case X of
e -> case Y of
  e -> {ok, L};
  {var, V} -> {fail, []}
  end;
{var, V} -> case Y of
  {var, V} -> {ok, L};
  {var, _} -> {ok, dd(lists:append(subList(V, Y, L), [V, Y]))};
  end;
{L, U} -> case Y of
  {var, V} -> unify_index(Y, X, L);
  {L, V} -> unify_index(U, V, L);
  end;
{r, U} -> case Y of
  {var, V} -> unify_index(Y, X, L);
  {r, V} -> unify_index(U, V, L);
  end;
{p, P1, P2} -> case Y of
  {var, .} -> unify_index(Y, X, L);
  {p, Q1, Q2} -> {Flag1, R1} = unify_index(P1, Q1, L),
  if (Flag1==ok) -> {Flag2, R2} =
  unify_index(subListTerm(R1, P2), subListTerm(R1, Q2), R1),
  if (Flag2==ok) -> {
    ok, R2};
  true -> {fail, []}
  };
true -> {fail, []}
end;
end.
```

As you may notice, indexes in type reconstruction unify in an ad-hoc way: variables unify only with other variables.
7.5 The other way round: from partial involutions to types

Sometimes, we need to delete already processed rules from a list \( P \); the following function precisely deletes rule \( R_1 \rightarrow R_2 \) from \( P \).

```haskell
delete_rule(R1, R2, P) ->
  case P of
    [] -> [];
    [R1, R2] | P1 -> P1;
    R | P1 -> R | delete_rule(R1, R2, P1)
end.
```

The following function allows one to check if a term \( R \) does not contain any occurrences of the constructors \( l \) and \( r \).

```haskell
lr_free(R) ->
  case R of
    e -> true;
    {var, _} -> true;
    {p, R1, R2} -> lr_free(R1) and lr_free(R2);
    {l, _} -> false;
    {r, _} -> false
end.
```

In order to convert to a plain readable string the internal representation of a partial involution term, we can use the following code:

```haskell
term2string(T) ->
  case T of
    e -> "e";
    {var, V} -> V;
    {l, T1} -> string:concat(string:concat("l ( ", term2string(T1) ), " )")
    {r, T1} -> string:concat(string:concat("r ( ", term2string(T1) ), " )")
    {p, T1, T2} -> string:concat(string:concat(string:concat(string:concat("< ", term2string(T1) ), ", "), term2string(T2), ">")
end.
```

Later on, during the type reconstruction process, we will need to scan indexes in order to search for occurrences of the constructors \( l \) and \( r \), in order to build, in a syntax driven way, the tree of type intersections. The following function is precisely devoted to such goal:

```haskell
leftmost_lr(R) ->
  String_R=term2string(R),
  %io:format("String_R: "p’\n", [String_R]),
  Match_L=string:str(String_R,"l"),
  Match_R=string:str(String_R,"r"),
  if Match_L=0 -> if Match_R=0 -> {not_found, none};
true -> New_String_R=lists:flatten(string:replace(String_R,"r","")),
%io:format("New_String_R: "p’\n", [New_String_R]),
```

26
The following function builds a type by recursion on the partial involution term \( T \), choosing \( \Omega \) placeholders fresh w.r.t. to variables in the list \( \text{Vars} \).

```java
build_type_tree(T, Vars) ->
    case T of
    e -> omega;
    [v, V] -> {var, V} |
    {l, T1} -> \( \Omega = \text{fresh}(\text{Vars} \cup \{\"\Omega\\}) \),
                map, build_type_tree(T1, \{Omega \mid Vars\}, \{\var, Omega\});
    {r, T1} -> \( \Omega = \text{fresh}(\text{Vars} \cup \{\"\Omega\\}) \),
                map, \{\var, Omega\}, build_type_tree(T1, \{Omega \mid Vars\});
    {p, I, T1} -> \( \Omega = \text{fresh}(\text{Vars} \cup \{\"\Omega\\}) \),
                  Check = true -> if
                  \( \text{Check} \neq true \) -> {bang, I, build_type_tree(T1, Vars)};
                  true -> {\text{Constr, Residual} = \text{leftmost}_lr(I)};
                  \( \text{Constr} \neq \text{cap} \) -> {\text{cap, build_type_tree}\{p, \text{Residual}, T1\},
                  \{\var, Omega\}};
                  \( \text{Constr} = \text{cap} \) -> {\text{cap, Omega}, build_type_tree\{p,\n                  \text{Residual}, T1\}, \{\var, \text{Vars}\}};
                  true -> {\text{var, Omega}}
    end
end.
```

The function \( \pi2type() \) uses \( \text{build_type_tree}() \) and unification algorithms on types and indexes, in order to map partial involution terms to their corresponding types.

```java
\( \pi2type(P,T) \) ->
    case P of
    [] -> T;
    [R1, R2 | P1] -> \( \text{R1-vars} = \text{vars}(R1) \),
                    \( \text{R2-vars} = \text{vars}(R2) \),
                    \( \text{P-vars} = \text{ListVars}(P) \),
                    \text{Subst1 = separateVars(\text{dd}(\text{R1-vars} ++ \text{R2-vars}), \text{P-vars}, \"X\")},
                    \text{FreshR1 = subListTerm(Subst1, R1)};
```

27
Since, the type unification processes carried out during the type reconstruction may leave out some useless "holes", represented by uninstantiated \( \Omega \) type variables, the following function allow us to scan the final reconstructed type and strip off such variables.

\[
\text{strip_extra_omega_variables} \quad (\text{Type}) \\
\begin{aligned}
\text{case Type of} \\
\quad \text{omega} \rightarrow \text{omega}; \\
\quad \text{map} \rightarrow \text{map}, \text{strip_extra_omega_variables}(\text{T1}), \text{strip_extra_omega_variables}(\text{T2}); \\
\quad \text{bang} \rightarrow \text{bang}, \text{strip_extra_omega_variables}(\text{T1}); \\
\quad \text{cap} \rightarrow \text{case T1 of} \\
\quad \quad \text{var}, X \rightarrow \text{PrefixCheckX} = \text{string : str(X, \"\Omega\")}, \\
\quad \quad \text{true} \rightarrow \text{case T2 of} \\
\quad \quad \quad \text{var}, Y \rightarrow \text{PrefixCheckY} = \text{string : str(Y, \"\Omega\")}, \\
\quad \quad \quad \text{true} \rightarrow \text{case T1 of} \\
\quad \quad \quad \quad \text{var}, Y \rightarrow \text{PrefixCheckY} = \text{string : str(Y, \"\Omega\")}, \\
\quad \quad \quad \quad \text{true} \rightarrow \text{case T2 of} \\
\quad \quad \quad \quad \quad \text{var}, Y \rightarrow \text{PrefixCheckY} = \text{string : str(Y, \"\Omega\")}, \\
\quad \quad \quad \quad \quad \text{true} \rightarrow \text{case T1 of} \\
\quad \quad \quad \quad \quad \quad \text{var}, Y \rightarrow \text{PrefixCheckY} = \text{string : str(Y, \"\Omega\")}, \\
\end{aligned}
\]

Since the type synthesis process depends on the order the partial involution rules are processed with, we make use of the following function to consider all the possible orderings:

\[
\text{permutations(L)} \\
\begin{aligned}
\text{case L of} \\
\quad [ ] \rightarrow [ ]; \\
\quad [\star, \] \rightarrow [ \star \mid H \mid T] \mid H \leftarrow L, T \leftarrow \text{permutations(L—H)};
\end{aligned}
\]
Finally, the user front-end function \texttt{show_principal_type()} simply calls the previous procedures and the auxiliary function \texttt{print_types()} below, in order to calculate and pretty print the principal type associated to the given partial involution (passed as a string).

\begin{verbatim}
show_principal_type(String) ->
  {ok,Tokens,1}=abramsky_lexer:string(String),
  {ok,P}=abramsky_parser:parse(Tokens),
  P1=check_dup(P),
  Perms=permutations(P1),
  print_types(Perms).

print_types(Perms) ->
case Perms of
  [] -> io:format("");
  [L|T] -> L1=dup(L),
          Type=pi2type(L1,[var,"Omega"]),
          FinalType=strip_extra_omega_variables(Type),
          if FinalType==[var,"Omega"] -> print_types(T);
          true -> pretty_print_type(FinalType),
           io:format("\n")
  end
end.
\end{verbatim}

\section{Putting all together}

For the sake of user’s convenience, we also provide a “catch-all” function, taking as a parameter the string representation of a $\lambda$-term and returning both the corresponding partial involution and principal type.

\begin{verbatim}
lambda2type(LambdaString) ->
  Combinators=export_combinators(),
  {ok,Tokens,1}=lambda_lexer:string(LambdaString),
  {ok,T}=lambda_parser:parse(Tokens),
  Comb_T=abstract(T),
  io:format("Combinators abstraction:\n"),
  pretty_print_lambda(Comb_T),
  io:format("\n"),
  Result=polish(decode(Comb_T,Combinators)),
  io:format("Partial involution rules:\n"),
  io:format("----- begin -----\n"),
  pretty_print_rules(Result),
  P1=check_dup(Result),
  Perms=permutations(P1),
  io:format("Principal type:\n"),
  print_types(Perms).
\end{verbatim}
References


