PrologPF: Function in Prolog

Ian Lewis,

University of Cambridge,

7th June 2012

Overview

PrologPF is a language that combines the logic programming of the language Prolog with the functional evaluation of the language ML. The syntax has been pragmatically designed to provide a consistent combined style that would be familiar to both Prolog and ML programmers.

Support for PrologPF is provided within any ISO-standard Prolog environment (development was within GNU Prolog) although similar support could be provided within a Standard ML environment.

Function *declaration* is supported with the `fun` keyword, e.g.

```
fun fact(X) =     if (X=1)
                  then 1
                  else X*fact(X-1).
```

The support for functions is provided within a Prolog environment, and function *evaluation* is supported through the use of declared functions in arguments to relations, e.g. a function `fact` can be used within a relation `map_fact`:

```
map_fact([],[]).
map_fact([H|T],[X|Y]) :- X = fact(H), map_fact(T,Y).
```

Functions can call relations within the condition element of an if-then-else statement, e.g.

fun f(X) = if (map_fact(X,R)) then R.

Higher-order functions

5.1 Introduction

The earlier implementations of parallel Prolog exploiting the Delphi principle, described in [28, 25, 76, 49, 66], can support programs written in a pure subset of Prolog. The use of the extra-logical predicate cut must be avoided, as was discussed in Chapter 4.

PrologPF extends the Delphi Machine to allow the use of cut, but only for deterministic procedures. The programmer must avoid the intentional or accidental use of cut within procedures which still (in spite of the cut) have multiple solutions.

However, the need for cut within a PrologPF program is greatly reduced as support is included for the definition and application of functions, in which the deterministic execution is ensured by the system. Also, Boolean functions can often be used where Prolog would rely upon the use of failure to express negation.

The higher-order functional support in PrologPF is sufficient to allow straightforward programming of all the exercises in an undergraduate ML functional programming course [60], and to allow a version of the SRI Prolog Technology Theorem Prover [70] to be implemented without cuts. The application of PrologPF to the functional programming exercises and PTTP is discussed in detail in Chapter 6.

PrologPF extends Prolog with support of the definition and deterministic evaluation of higher-order functions, with the functions treated as first-class values within the logic system. The Delphi oracles do not extend into the functional reduction graph, and no parallelism is provided for the evaluation of an individual function call. This is consistent with the objective of replacing Prolog procedures containing cuts. PrologPF does not attempt to exploit all the parallelism available in the non-deterministic but complete evaluation of functions treated as general equational theories using algorithms such as lazy narrowing. Chakravarty and Lock provide the semantics and an implementation of lazy narrowing in [20].

While PrologPF provides a consistent environment for higher-order functional programming, the language has the same syntax (with the definition of some additional operators) as normal Prolog. Thus a PrologPF program can be read by a standard Prolog compiler to produce a program in which all function applications are treated as irreducible Prolog terms.

By careful selection of the specially treated operators, the functional syntax of PrologPF will be familiar to users of Standard ML.

5.1.1 Implementation goals

1. To be compatible with the Delphi principle, functional reduction must be deterministic
2. The capabilities of the functional component of PrologPF should minimise the requirement for cut in the body of Prolog rules
3. The syntax should allow functional algorithms to be clearly expressed, with support for Prolog terms and variables including those representing functions, i.e. higher-order functions should be supported

4. The syntax and semantics of PrologPF should facilitate the straightforward use of functions within Prolog rules, and permit deterministic calls to Prolog procedures from within functions

5.2 Function definition: the fun relation

5.2.1 A PrologPF example

Before reviewing the syntax and semantics of PrologPF functions in detail with comparison to other approaches, the following examples of the factorial and append functions in PrologPF may place the alternatives in context.

Firstly, the factorial function:

```
fun   fact(1) = 1;
      fact(N) = N * fact(N-1).
```

or equally (see Section 5.5.4):

```
fun fact(N) =     if (N = 1)
                  then 1
                  else N * fact(N-1).
```

The append function can be defined as follows:

```
fun   append( [],Y)     = Y;
      append([X|Xs],Y)  = [X|append(Xs,Y)].
```

5.2.2 The PrologPF approach

Functions are defined in PrologPF with the special relation `fun/1`, which is defined as a Prolog prefix operator of low precedence with `op(1200,fx,fun)`.

Function definition in PrologPF also uses the `=` and `;` operators but the standard Prolog precedence has been maintained.

The syntax supported is shown in Table 5.1

In PrologPF, the underlying Delphi Machine has been extended to support cut (see Chapter 4), and this support is exploited to implement deterministic functional reduction.

Each fun relation is transformed through a process of *flattening* [22] into a deterministic procedure, with the actual arguments being matched against the formal parameters until a successful unification is made, at which point the choice of equality rule is committed and the reduction continuing with the term on the right-hand-side. Thus the selection of the appropriate equality rule is top-down, and the rewrite is strictly left-to-right.

The equality is required to be *constructor-based*, that is the terms in the function head must not themselves contain any defined functions. This requirement is also described as *head normal*

form [42]. The syntax of the formal parameters is given in Table 5.1 as Prolog_Term, i.e. a standard Prolog term not including the application of any defined functions.

While the operational semantics of function evaluation in PrologPF have most in common with languages such as Standard ML [61, 55], the argumentmatching process is replaced with Prolog's unification. Argument unification in PrologPF thus differs from the matching in functional languages such as ML in two significant ways:

1. There is no requirement for left-linearity in the equality rules, i.e. variables can be repeated in the function head. The functional component of PrologPF, like the underlying Prolog, has no occurs check. As with Prolog, it is the programmer's responsibility to avoid actual parameters which would cause the unification algorithm to loop, as with the goal `:- Y = a(Y).`
2. Partially instantiated data structures (i.e. terms containing logical variables) can be passed as arguments and returned as results. This means that, for example, difference lists can be supported and that a list of variables can be appended to another.

The Prolog atom used to name a defined function denotes a function of fixed arity, set by the number of formal parameters given in the `fun` relation. Alternative definition of functions using the same name but a differing number of parameters is flagged as an error by the PrologPF compiler. This approach clearly differs from the Prolog style where a relation name can be considered a combination of the naming atom and the arity (as in `foo/2`), but is essential to permit currying within the standard Prolog syntax.

Function_Definition	::=	**fun** Alternate_Definitions **.**
Alternate_Definitions	::=	Fun_Equality | Fun_Equality **;** Alternate_Definitions
Fun_Equality	::=	Fun_Head **=** PrologPF_Term
Fun_Head	::=	Prolog_Atom **(** Args… **)** | Prolog_Atom **@ [** Args… **]** | Prolog_Atom **@ []**
Args	::=	Prolog_Term | Prolog_Term **,** Args
PrologPF_Term	::=	Prolog_Term | Function_Application

Table 5.1: Syntax: Function Definition with the `fun` Relation

5.2.3 Alternative approaches

5.2.3.1 Deterministic relations in Prolog

Within Prolog, it is possible to define deterministic relations which then can be treated as functions:

```
fact(1,1).
fact(N,F) :- N > 1, N1 is N - 1, fact(N1,F1), F is N * F1.
```

In general, however, determinism inference is an undecidable problem, at least dependent upon the solution of the halting problem:

```
foo(X,Y) :- complicated(X,Y).
foo(X,X).
```

`foo/2` can have more than one solution only if `complicated/2` can succeed.

In many cases, the programmer uses cut within the Prolog program to ensure determinacy of an otherwise non-deterministic relation. For example:

```
fact(1,1) :- !.
fact(N,F) :- N1 is N - 1, fact(N1,F1), F is N * F1.
```

However, the presence of cut is not enough to guarantee determinacy, as in the following example:

```
a(a).
a(b) :- !.
a(c).
```

The query `:-a(X).` has the multiple solutions `X=a, X=b`.

Deterministic reduction is essential for the successful support of functions on the Delphi Machine (see Chapter 4), so the use of un-annotated Prolog relations to define functions would introduce a significant possibility of error.

5.2.3.2 Mercury

In the Mercury system, each procedure is annotated with determinism information [43]. The syntax of Prolog relation definition permits the use of relations and functions in multiple modes, i.e. differing arguments being instantiated at the time of the call, with others expected as results. Mercury functions are thus annotated with determinism information for each mode.

For example:

```
:- pred factorial(int, int).
:- mode factorial(in,out) is det.
factorial(N, F) :-
            ( N =< 0 ->
                  F = 1
            ;
                  N1 is N - 1,
                  factorial(N1, F1),
                  F is F1 * N
```

```
        ).
```

Note that the mode information defines `factorial` to be `det`, i.e. deterministic, while the relational style of definition is retained. The Mercury compiler checks the supplied determinism information by analysis of the code. In this example the alternative representation of the function shown below would be inferred to be non-deterministic through limitations in the compiler's analysis of mutually exclusive conditions, so the earlier if-then-else form must be used:

```
factorial(0, 1).
factorial(N, F) :-
        N > 0,
        N1 is N - 1,
        factorial(N1, F1),
        F is F1 * N.
```

The use of Mercury's determinism and type inferencing techniques have potential for exploitation on the Delphi Machine. In PrologPF all functions are, to use Mercury terminology, semi-deterministic. That is they can succeed once or fail. The issue of function failure in PrologPF is discussed in Section 5.7. Non-deterministic modes of functions are not required, and the syntax of function definition and application can be considerably simplified and optimised for the deterministic use.

5.2.3.3 Curry

The logic capabilities of the language Curry [42] are provided through the support for *non-deterministic functions*, and the function definition syntax supports this:

```
f :: Int -> Int
f 1 = 10
f 2 = 20
f 2 = 30
```

The language is typed, with `f` defined as *int* → *int* above. The call `f 2` will produce the multiple results 20 and 30. The left-hand-sides of the functional equality definitions can be defined with conditional guards, such that the definitions are referred to as conditional equations where the conditions are constraints which must be solved in order for the equation to be applied.

This form is used in the definition of factorial:

```
factorial :: Int -> Int
factorial 1             = 1
factorial n | n > 1     = n * factorial (n - 1)
```

The constraint `n > 1` is added to the second equality defining the factorial function to ensure deterministic evaluation of factorial 1 which would otherwise match the right-hand side of both rules. To ensure deterministic execution of a function in Curry, the defining equations must be checked to ensure that the conditions are not simultaneously satisfiable [56], and no new variables can be introduced in the equations' right-hand sides.

The condition constraint in Curry can also be a boolean function expression, as an abbreviation for the rule `<bool expr>=True`. This is similar to the treatment of function applications in relation positions in PrologPF, discussed in Section 5.6.

5.2.3.4 External procedures

The functions can be defined in a language other than Prolog, and called as external procedures. Many existing implementations of Prolog support this capability, and effort has been made to formalise the approach [14, 53, 13]. These systems do not support higher-order programming.

5.2.3.5 Logic programming with equality

A more general solution is to define functions in terms of a set of equalities [41, 57], extending Prolog's '=' relation, with conditional support provided in the form of *guards*. For example:

```
fact(1) = 1.
fact(N) = N * fact(N-1) :- N \== 1.
```

The use of guards (in this example `N \== 1`) provides access to Prolog relations, including those with multiple solutions. The use of the equality relation itself imposes no constraints on the form of the definition, permitting for example

```
append(X,append(Y,Z)) = append(append(X,Y),Z).
```

This is useful if a most general equation solving procedure is to be used, with non-deterministic selection of rewrite rules and of terms for reduction, and right-to-left as well as left-to-right application of each equality rule.

The non-deterministic solution of equations would provide interesting opportunities for the application of the Delphi principle to the extended proof tree. However, the research in this dissertation ensures the functional reduction process is deterministic such that the parallelised program has the efficiency associated with direct execution of compiled machine code.

5.3 Function application: the @ operator

The development of the `@` operator as a relation denoting function application in Prolog, with an interpretation expressed in Prolog, can be found in [27].

5.3.1 Extending Prolog for explicit function application

The standard syntax for Prolog terms is supported, with special meaning applied to a new operator @ (defined in PrologPF as `op(600,yfx,@)`). The presence of the operator in a PrologPF term indicates that the normal unification step should be preceded by functional evaluation.

For example, in the goal for the relation "=":

```
:- Z = foo @ [a].
```

the term `foo @ [a]` should be evaluated before the terms `Z` and the result of `foo @ [a]` are unified with the arguments of the = relation.

If foo is a *defined function* (i.e. defined with the fun relation described in Section 5.2), then the rewrite rules specified in the associated fun relation are used for the reduction. Otherwise foo is a constructor and the term is irreducible.

For nested `@` terms, function evaluation is *strict*, i.e. innermost arguments are evaluated first. For example in:

```
:- Z = foo @ [goo @ [a], hoo @ [b]].
```

the terms `goo @ [a]` and `hoo @ [b]` will be evaluated before the results are used in the evaluation of foo with those arguments. The evaluation of argument terms takes place left-to-right. Evaluation ordering is significant in PrologPF because the usual functional programming one-way *matching* is replaced with *unification*, and variable arguments are permitted. The full `@` syntax is given in Table 5.2.

Function_Application	::=	Function_Term **@ [** Args… **]** | Function_Term **@ []** | Defined_Atom **(** Args… **)**
Function_Term	::=	Defined_Atom | Variable | Lambda_Expression | Function_Application
Lambda_Expression	::=	**lambda([**Formal_Args…**],** PrologPF_Term**)** | **lambda([],** PrologPF_Term**)**
Formal_Args…	::=	Prolog_Term | Prolog_Term **,** Formal_Args…
Args…	::=	PrologPF_Term | PrologPF_Term **,** Args…
Defined_Atom	::=	Prolog_Atom defined in earlier `fun` claus

Table 5.2: Syntax: Function Application with the `@` Operator

Note that a function is always applied to a list of arguments, so terms such as `foo @ a` or `foo @ X` do not denote function application (the correct syntax would be `foo @ [a]` and `foo @ [X]`).

A function `foo` can be defined with no arguments, and the reduction of that function can be made explicit with `foo @ []`. This use of *nil* is similar to the value *unit* in Standard ML, and is useful where function abstractions are used to emulate laziness, as in the example with infinite lists in Chapter 6. Nil argument functions are discussed further in Section 5.8.

5.3.2 Function application: syntactic sugaring

It should be noted that in PrologPF the term:

```
foo(a,b)
```

in which foo is a defined function, is semantically equivalent to:

```
foo @ [a,b]
```

This allows the most convenient syntax for function application to be used within PrologPF programs and allows consistent treatment of constructors and functions. For example, the solution of the goal:

```
:- Z = foo(goo(a),hoo(b)).
```

can involve functional reduction of any of `foo`, `goo`, or `hoo`. With `fun goo(X) = gg.` and `fun hoo(X) = hh.` then the goal will succeed with the single solution `Z = foo(gg,hh).`

This consistent treatment of constructors and functions can be seen in the definition of a `wrap` function which maps a list to a similar list with each element wrapped with the constructor `envelope`:

```
fun   wrap([])    = [];
      wrap([X|T])  = [envelope(X)|wrap(T)].
```

5.4 Higher-order functions and currying

A goal of the PrologPF system is to support functions as first-class data items in the extended Prolog semantics, and to permit a syntax which facilitates the straightforward creation and application of function closures.

The approach in PrologPF owes much to Standard ML [55], with support for nameless functions as lambda-expressions and the creation of closures via currying [33, 67].

5.4.1 Lambda-expressions

Nameless functions are created in PrologPF using the special constructor `lambda/2`. The syntax is given in Table 5.2.

An example of a goal using a lambda expression representing the increment function is:

```
:- Z = lambda([X],X+1) @ [6].
```

returning the single solution `Z = 7`.

As with defined functions in PrologPF, the evaluation of the function term proceeds with the unification of the actual parameter (in this example 6) with the argument of the lambda expression (`X`). The instantiated second argument of the `lambda` term is then evaluated to produce the final result.

Unlike standard Prolog, the scope of the formal arguments of the lambda expression (X in the example above) is limited to that expression. This ensures the correct operation of goals such as:

```
:- Y = lambda([X],X+1) @ [6], Z = lambda([X],X*2) @ [7].
```

PrologPF lambda terms can be defined to take **no** arguments, providing a mechanism to delay the evaluation of the expression given as the secondargument. For example:

```
Z = lambda([],f(100))
```

The expression `f(100)` will not be evaluated until a subsequent application `Z @ []`. This use of *nil* arguments is discussed further in Section 5.8.

5.4.2 Currying

The support for currying in PrologPF ensures that the following equivalence holds true:

```
foo @ [a] @ [b] @ [c] ≡ foo @ [a,b,c]
```

The arity of a defined function is fixed in the `fun` relation (Section 5.2). Any alternate definition using the same function name but with a differing number of formal parameters is flagged by PrologPF as an error. This means the PrologPF compiler can generate appropriate code to return a lambda expression where a function is called with fewer arguments than appear in the `fun` definition. The definition of the operator `@` was shown in Section 5.3 to be left-associative (the 'yfx' in `op(600,yfx,@)`.

These capabilities combine to provide the flexible support for higher-order abstraction through the partial application of functions, known as currying.

For example, if a function foo is defined with 3 arguments as in:

```
fun foo(X,Y,Z) = X+Y+Z.
```

then (using symbol → to represent 'evaluates to'):

```
foo @ [a] → lambda([Y,Z],foo(a,Y,Z))

=>

foo @ [a] @ [b] @ [c]   ≡ ((foo @ [a]) @ [b]) @ [c]
                        → (lambda([Y,Z],foo(a,Y,Z)) @ [b]) @ [c]
                        → lambda([Z],foo(a,b,Z)) @ [c]
                        → foo(a,b,c)
                        ≡ foo @ [a,b,c]
```

The explicit use of the `@` operator and the use of currying permit the straightforward definition and application of functions such a map:

```
fun   map(F,[]) = [];                    % map definition
```

```
        map(F,[X|Xs]) = [F @ [X]|map(F,Xs)].
:- Z = map(+1,[10,20,30]).                % curried +
:- Inc = map(+1), Z = Inc @ [[10,20,30]]. % curried map, +
```

Each query succeeds with the single solution for `Z = [11,21,31]`.

5.5 Special treatment of if-then-else

PrologPF includes a predefined function if to provide conditional evaluation of alternative expressions. The systematic eager evaluation in PrologPF precludes the definition of if as a normal PrologPF function with three arguments:

```
fun   if(true, A,B) = A;
      if(false,A,B) = B.
```

As the argument evaluation semantics of PrologPF are eager, in an expression such as `if(Z=0, 1, 100/Z)` all three arguments would be evaluated before the application of if, producing a possible run-time arithmetic error during the attempted evaluation of `100/Z`.

To provide more useful behaviour, if is treated as a predefined function with exceptional semantics. The special treatment is unique to `if`:

1. The evaluation of the alternative expressions is delayed until **after** the condition has determined which of the two alternatives should be evaluated. Only **one** of the two alternatives will then be evaluated.
2. The condition term is treated as a Prolog **goal**, rather than a boolean valued reducible expression.

5.5.1 Syntax

If_Expression	::=	**`if(`** PrologPF_Term$_1$, PrologPF_Term$_2$, PrologPF_Term$_3$ **`)`** | **`if`** PrologPF_Term$_1$ **`then`** PrologPF_Term$_2$ **`else`** PrologPF_Term$_3$ | **`if`** PrologPF_Term$_1$ **`then`** PrologPF_Term$_2$

Table 5.3: Syntax: `if`

The syntax for the conditional `if` expression is given in Table 5.3.

The use of the predefined operators `if`, `then` and `else` is permitted to reduce the use of brackets and allow a syntax similar to that of languages such as Standard ML. Where the `if-then-else` form is used, the resultant expression is equivalent to the term `if(Term1,Term2,Term3)`.

To allow a convenient syntax without modifying the precedence of the standard Prolog operators, the following precedences are used for `if`, `then` and `else`:

```
:- op(675,fx,if).       % 'if' is prefix
:- op(650,xfx,then).    % 'then' is infix
```

```
:- op(625,xfx,else).    % 'else' is infix
```

The precedence of the predefined if, then and else operators in PrologPF implies that:

```
if Term1 then Term2 else Term3
≡ if (Term1 then (Term2 else Term3 ))
≡ if(then(Term1,else(Term2,Term3)))
```

The `else`-expression can be omitted, such that:

```
if Term1 then Term2 ≡ if Term1 then Term2 else fail
```

The precedence of the `if-then-else` compound term has been set higher than that of the Prolog's = and ; operators to minimise the need for brackets in function definitions, and in goals of the form `Z = if-expression.` The compromise means that conditional operators used in `if` conditions (i.e. $Term_1$) must be bracketed, as must be nested `if` expressions.

For example:

```
if (A < 20) then (if (A > 12)
                  then middle
                  else lower
                  )
               else upper
```

5.5.2 Evaluation

Special code is generated in the call to if in the evaluation of `if`-expressions.

5.5.2.1 Defined evaluation ordering with if

For any other arity/3 function call such as `foo(Term1,Term2,Term3)` for

defined function `foo`, code of the following form would be generated:

[code to evaluate $Term_1$ with result as term X_1]

[code to evaluate $Term_2$ with result as term X_2]

[code to evaluate $Term_3$ with result as term X_3]

functional evaluation of `foo(X1,X2,X3)`

In the case of the special function `if` the eager evaluation of both alternative expressions in terms such as `if (Z = 0) then 1 else 100/Z` would not execute as intended for `Z = 0`, so consequently code of the following form will be generated:

[code to find first solution of `call`(*$Term_1$*) *as relational goal]* (Section 5.5.2.2)

<on success:> *[code to return result of evaluation of $Term_2$]*

<on failure:> *[code to return result of evaluation of Term$_3$]*

PrologPF ensures that:

1. The condition goal completes **before** the evaluation of the alternate expressions of the `if`-expression.
2. The condition goal succeeds with one solution, or fails.
3. Only one of the alternate expressions will be evaluated: the `then`-expression if the condition goal succeeds, or the `else`-expression if it fails.

5.5.2.2 `if` condition as relational goal

There is considerable advantage in giving functions within the combined functional logic system access to the relations in the program and those in the Prolog libraries. The implementation chosen for the Delphi Machine requires that the function evaluation be deterministic. A successful compromise has been achieved with:

1. The **only** place a function in PrologPF can call a Prolog relation is in the condition of an `if`-expression
2. The call uses Prolog's normal search, but determinism is maintained with first-solution semantics
3. The acceptance of boolean functions as relational goals reintroduces functional terms as conditions (Section 5.6)

An example showing how the Prolog library append relation can be used to produce a similar (but deterministic) function would be:

```
fun append(X,Y) = if append(X,Y,Z) then Z.
```

This example relies upon the following:

1. The if semantics ensure the goal append produces a value for Z before the evaluation of the sub-expressions Z and fail.
2. The if semantics ensure that only one of the sub-expressions is evaluated, after the solution of the conditional goal.
3. The relation append/3 and the function append/2 are recognised as having different names (see Section 5.6)
4. The missing else-expression is equivalent to else fail, so the definition is an abbreviation for:
   ```
   fun append(X,Y) = if append(X,Y,Z) then Z else fail.
   ```
5. The predefined function fail is available to produce function failure (Section 5.7)

The use of relational goals as conditions, combined with Prolog's left-to-right search rule, leads to a Prolog syntax with semantics similar to the special operators in languages such as Standard ML for `andalso` and `orelse` [55]:

Conjunction: (P,Q,R) ≡ `P andalso Q andalso R`

Disjunction: (P;Q;R) ≡ `P orelse Q orelse R`

For example, using the standard Prolog library relations > and <:

```
fun account_status(Bal) =     if (Bal > 0, Bal < 100)
                              then normal
                              else needs_attention.
```

In using a relational goal as the condition, the PrologPF if expression has similar behaviour to the Prolog conditional goal, written `A -> B; C`. The definition of the operators "`->`" and "`;`" are provide in [35]. The subgoal `A` is called to provide one solution or fail. In the former case, subgoal `B` is then called, else subgoal `C` is called. The semantics are complicated by the presence of any cuts in subgoals `A`, `B` or `C`. The deterministic execution of functions in PrologPF permits the provision of an *if-then-else* expression without these complexities.

5.5.3 Value declarations

A value declaration gives an expression a *name* within a particular *scope*.

The PrologPF support for `if` ensures that the relational condition is executed before the alternate expressions. The unifier of the free variables in the condition is thus valid for the evaluation of the `then`-expression, which is only evaluated if the condition has succeeded. Thus the use of the = relation in the condition of an `if-then-else` expression can give a value a name, which will be valid in the scope of the `then` sub-expression.

The use of the unification of the condition to support naming in this way is convenient if a sub-expression is to be repeated within an expression, as often occurs within an `if-then-else`. An example is in a definition of a max function to find the highest integer in a list:

```
fun     max([X])        =       X;
        max([X|Xs])     =       if (M = max(Xs))
                                then (  if (X > M)
                                        then X
                                        else M
                                     ).
```

In the recursive case, the condition goal `M = max(Xs)` results in the evaluation of `max(Xs)` being unified with a new free variable `M`, with the unifier `M/n` (where `n` is the largest integer in `Xs`) being valid for the subsequent evaluation of `if (X > M) then X else M`.