FProlog, adding functions to Prolog via a mainstream approach

Ian J. Lewis

February 21, 2019

Abstract

Most approaches to adding functions to Prolog preserve the relational clause syntax with additional output arguments intended to hold the functional results. This paper argues for a syntax much more sympathetic to functional programming such that functional code in FProlog would be familiar to ML programmers and existing ML programs can be easily ported to FProlog.

1 Introduction

The earlier implementations of parallel Prolog exploiting the Delphi principle, described in [8, 6, 28, 14, 24], 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 (Cut Chapter ref).

Fprolog 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 Fprolog 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 Fprolog is sufficient to allow straightforward programming of all the exercises in an undergraduate ML functional programming course [21], and to allow a version of the SRI Prolog Technology Theorem Prover [27] to be implemented without *cuts*. The application of Fprolog to the functional programming exercises and PTTP is discussed in detail in Chapter (Case Studies Chapter Ref).

Fprolog 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*. Fprolog 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 [4].

While Fprolog 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 Fprolog 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 Fprolog will be familiar to users of Standard ML.

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 Fprolog 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 Fprolog should facilitate the straightforward use of functions within Prolog rules, and permit deterministic calls to Prolog procedures from within functions

2 Function definition: the `fun` relation

2.1 A Fprolog example

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

Firstly, the factorial function can be defined in Fprolog using alternate head clauses as below:

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

or equally the familiar if-then-else syntax can be used (see Section 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)].
```

2.2 The Fprolog approach

Functions are defined in Fprolog 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 Fprolog also uses the `=` and `;` operators but the standard Prolog precedence has been maintained.

The syntax supported is shown in Table 1

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

Each `fun` relation is transformed through a process of *flattening* [5] 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* [12]. The syntax of the formal parameters is given in Table 1 as Prolog_Term, i.e. a standard Prolog term not including the application of any defined functions.

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

Table 1: Syntax: Function Definition with the `fun` Relation

While the operational semantics of function evaluation in Fprolog have most in common with languages such as Standard ML [22, 17], the argument matching process is replaced with Prolog's *unification*. Argument unification in Fprolog 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 Fprolog, 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 Fprolog 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.

2.3 Alternative approaches

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 (Cut Chapter ref)), so the use of unannotated Prolog relations to define functions would introduce a significant possibility of error.

2.3.2 Mercury

In the Mercury system, each procedure is annotated with determinism information [13]. 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 Fprolog all functions are, to use Mercury terminology, semi-deterministic. That is they can succeed once or fail. The issue of function failure in Fprolog is discussed in Section 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.

2.3.3 Curry

The logic capabilities of the language Curry [12] 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 $int \rightarrow int$ 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 [18], 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 Fprolog, discussed in Section 6.

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 [3, 16, 2]. These systems do not support higher-order programming.

2.3.5 Logic programming with equality

A more general solution is to define functions in terms of a set of equalities [11, 19], 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.

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 [7].

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 Fprolog as `op(600,yfx,@)`). The presence of the operator in a Fprolog 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 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 Fprolog because the usual functional programming one-way *matching* is replaced with *unification*, and variable arguments are permitted. The full `@` syntax is given in Table 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... `]` `,` Fprolog_Term `)`
	|	`lambda([],` Fprolog_Term `)`
Formal_Args...	::=	Prolog_Term
	|	Prolog_Term `,` Formal_Args...
Args...	::=	Fprolog_Term
	|	Fprolog_Term `,` Args...
Defined_Atom	::=	Prolog_Atom defined in earlier `fun` clause

Table 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 (Case Studies Chapter Ref). Nil argument functions are discussed further in Section 8.

3.2 Function application: syntactic sugaring

It should be noted that in Fprolog 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 Fprolog 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)].
```

4 Higher-order functions and currying

A goal of the Fprolog 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 Fprolog owes much to Standard ML [17], with support for nameless functions as lambda-expressions and the creation of closures via currying [9, 25].

4.1 Lambda-expressions

Nameless functions are created in Fprolog using the special constructor `lambda/2`. The syntax is given in Table 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 Fprolog, 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].
```

Fprolog lambda terms can be defined to take **no** arguments, providing a mechanism to delay the evaluation of the expression given as the second argument. 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 8.

4.2 Currying

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

`foo @ [a] @ [b] @ [c]` $\equiv$ `foo @ [a,b,c]`

The arity of a defined function is fixed in the `fun` relation (Section 2). Any alternate definition using the same function name but with a differing number of formal parameters is flagged by Fprolog as an error. This means the Fprolog 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 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 $\leadsto$ to represent 'evaluates to'):
`foo @ [a]` $\leadsto$ `lambda([Y,Z],foo(a,Y,Z))`
$\Longrightarrow$

```
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 Special treatment of `if-then-else`

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

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

As the argument evaluation semantics of Fprolog 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.

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

Table 3: Syntax: `if`

2. The condition term is treated as a Prolog **goal**, rather than a boolean-valued reducible expression

5.1 Syntax

The syntax for the conditional `if` expression is given in Table 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(`Term$_1$`,`Term$_2$`,`Term$_3$`)`.

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 Fprolog implies that:
`if` Term$_1$ `then` Term$_2$ `else` Term$_3$
$\equiv$ `if (`Term$_1$ `then (`Term$_2$ `else` Term$_3$ `))`
$\equiv$ `if(then(`Term$_1$,`else(`Term$_2$,Term$_3$`)))`

The `else`-expression can be omitted, such that:
`if` Term$_1$ `then` Term$_2$ $\equiv$ `if` Term$_1$ `then` Term$_2$ `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
```

5.2 Evaluation

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

5.2.1 Defined evaluation ordering with `if`

For any other arity/3 function call such as `foo(`Term_1`,`Term_2`,`Term_3`)` 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(`X_1,X_2,X_3`)`

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 5.2.2)
<on success:> *[code to return result of evaluation of* $Term_2$*]*
<on failure:> *[code to return result of evaluation of* $Term_3$*]*

Fprolog 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.

5.2.2 `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 Fprolog 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 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 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 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` [17]:

Conjunction:	`(P,Q,R)`	$\equiv$	`P andalso Q andalso R`
Disjunction:	`(P;Q;R)`	$\equiv$	`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 Fprolog `if` expression has similar behaviour to the Prolog conditional goal, written `A -> B; C`. The definition of the operators "`->`" and "`;`" are provide in [10]. 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 Fprolog permits the provision of an *if-then-else* expression without these complexities.

5.3 Value declarations

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

The Fprolog support for `if` 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`.

The use of `M` as a *name* to represent the value `max(Xs)` is equivalent to the repeated appearance of the value in the `then` expression. The `max` function could equally be written:

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

As these value declarations are using the standard `=` relation in the condition, the method supports a convenient technique for using functions that return multiple results as a tuple. This can be seen with the second of the complementary functions `zip` and `unzip`. The function `zip` takes two lists of equal length as arguments, and returns a list of pairs [12]:

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

The complementary function `unzip` has a convenient definition using a value declaration [22]:

```
fun unzip([])            = ([],[]);
    unzip([(X,Y)|Pairs]) = if ((Xs,Ys) = unzip(Pairs))
                           then ([X|Xs],[Y|Ys]).
```

A version of `unzip` that did not use a value declaration could be written using of auxiliary functions to extract the elements of the tuple and repeating the `unzip(Pairs)` sub-expression. Alternatively, an auxiliary function could be defined to add a pair of elements to pair of lists, as in:

```
fun addpair((X,Y),(Xs,Ys)) = ([X|Xs],[Y|Ys]).

fun unzip([])           = ([],[]);
    unzip([Pair|Pairs]) = addpair(Pair, unzip(Pairs)).
```

However, the use of value declarations in `unzip` avoids the use of auxiliary functions.

In the absence of common-expression elimination optimisations in the Fprolog compiler, the use of value declarations results in a more efficient object program. In general, the use of names to represent expressions that are complex or repeated can result in programs that are more comprehensible.

5.4 Alternate function definitions $\equiv$ `if`

In Fprolog, the function definition style using alternate argument patterns can be shown to be equivalent to a single functional equality using the predefined `if` function.

The following characteristics of the `if` function are important in the equivalence:

- The defined lazy conditional evaluation of the arguments to `if`
- The call to the condition goal is defined to *precede* the evaluation of one of the functional terms. Value declarations from unification of terms in the condition goal with local variables are therefore guaranteed to be bound in the scope of the subsequently evaluated dependent expression.

With the example of the factorial function:

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

The equivalent if-then-else form is:

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

The general form of the translation is:

```
fun Function_name(Arg_pattern1, Arg_pattern2...) = Expression_1;
    Function_name(Arg_pattern3, Arg_pattern4...) = Expression_2...
```

goes to:

```
fun Function_name(Var1,Var2...)
          = if (Var1 = Arg_pattern1, Var2 = Arg_pattern2...)
            then Expression_1
            else (if (Var1 = Arg_pattern3, Var2 = Arg_pattern4...)
                  then Expression_2
                  else ...
                 ).
```

Value declarations in the relational goal of the `if` condition can be seen more clearly with the transformation of the `append` function:

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

goes to:

```
fun append(L,Y) = if (L=[])
                  then Y
                  else (if (L=[X|Xs])
                        then [X|append(Xs,Y)]
                       ).
```

5.5 Summary of Fprolog `if` semantics

The goal of the `if-then-else` implementation in Fprolog is to provide useful conditional evaluation semantics, while supporting deterministic access to relations.

With the expression `if` Term_1 `then` Term_2 `else` Term_3:

- The conditional expression Term_1 is treated as a relational goal, either succeeding with a variable binding, or failing.
- The depth-first, left-to-right search of standard Prolog is used to find a solution to Term_1, and the search is limited to finding the first solution.
- The call to the conditional expression Term_1 completes before the evaluation of either Term_2 or Term_3.
- If Term_1 succeeds then Term_2 is evaluated in the context of any bindings resulting from the solution of Term_1, and the result returned as the value of the `if`-expression.
- If Term_1 fails then Term_3 is evaluated and returned as the result of the `if`-expression.
- If the `else`-expression (`else` Term_3) is omitted, the semantics are the same as if an `else`-expression (`else fail`) were added.

6 Boolean functions as relations

In summary, the following equivalence holds for functions used in the position of relational goals:

`?- foo(a).` $\equiv$ `?- foo(a) = true`

iff `foo` is a defined function of arity/1.

A function application term is permitted to appear in the position of a relational goal, where it is treated as a call to the Prolog `=` relation to unify the result of the function application with `true`. This applies to the body of each rule and the condition of each `if` expression.

For example, with a boolean function `prime(X)` returning true for a prime argument and false otherwise, the goal:

```
?- p(X), prime(X), write(X).
```

is equivalent to:

```
?- p(X), prime(X) = true, write(X).
```

The explicit treatment of boolean functions as relations in this way can be seen in the prototype produced by Paulson and Smith [23]. The language Escher [15] has **all** relations declared as boolean functions in this way.

Either the explicit `@` operator can be used to denote the function application, or the Prolog compound term syntax can be used. In the latter syntax, the outermost functor of the goal will only be recognised as a defined function if the number of actual parameters matches the arity of the defined function of the same name. The specification of a reduced number of arguments in a curried application is not useful where a boolean result is required. A partial (i.e. curried) function application would always return a higher-order result, such that:

(<curried_application> = true) $\equiv$ fail

The requirement for the arities of the defined function and the actual use within a goal facilitates the conversion of library relations (such as `append`) into functions and vice versa. I.e. the functional definition of `append/2` does not conflict with the relational definition `append/3`, and the library relation can be used in the function definition:

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

Equally, the deterministic functional version of append given in Section 5.4 could have been used for a version of the library relation limited to deterministic modes:

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

To summarise the naming/arity issues arising from both currying and the acceptance of boolean functions as relations:

1. Each alternate equality statement in the definition of a function must have the same number of formal parameters, and this number is the arity of the function.
2. A function can have the same name as a relation, but must not have the same arity.

The first rule is to allow currying, the second to allow boolean functions as goals. The functional logic language Mercury has a similar rule to 2 above, but in Mercury a function must not have an arity that is **one less** that a relation of the same name. The Mercury name/arity constraints are inconvenient, as it is natural to define a function (such as `append/2`) to have an arity one less that an equivalent relation (i.e. `append/3`). Fprolog exploits this capability to define functions representing deterministic modes of many frequently-used library relations such as `append` and `=..`.

In the design of Fprolog, a choice was made to introduce rule 2, rather than the alternative that boolean functions as goals should require explicit use of the `@` operator. The body of Prolog code converted for execution on the Fprolog system has not yet included enough examples of relations with multiple arities to confirm this design decision.

7 Failure of functions

The functional support in Fprolog is embedded within an environment of relations which are expected to `succeed` (with an associated variable binding) or *fail*. The treatment of function applications as relation argument terms associates every application with an underlying relation, for example in:

```
?- Z = fact(5).
```

the function application of `fact` is as an argument of the relation `=`.

7.1 Functional failure ⇒ Relation failure

In Fprolog, function failure is supported through the provision of a special term `fail`. This mirrors the standard Prolog relation `fail`, which always fails.

1. The evaluation of the term `fail` within an expression produces no value but always fails.

2. A function application fails if evaluation of a subexpression in the right-hand-side of the associated definition fails.

3. A relation fails if the evaluation of a functional argument fails.

The use of `fail` within a function definition can be seen in the `lookup` function, which returns a value associated with a key in a list of paired key-value terms:

```
fun lookup(_,[])                    = fail;
    lookup(Key,[(Key,Value)|_]) = Value;
    lookup(Key,[_|T])           = lookup(Key,T).
```

The function might be used in a program such as:

```
a(a).
a(b).
a(c).

?- a(X), write(lookup(X,[(a,1),(c,3),(e,5)])).
```

The subgoal `a(X)` produces values `a`, `b` and `c` for `X`, calling `write` with the value of the `lookup` application. As the key-value list argument contains no entry for `b`, the application will fail for that argument value. Backtracking will take place as in standard Prolog, such that `write` will display the values `1` and `3` from the successful application of `lookup` with `a` and `c`.

7.2 Function `fail` as an exception

Within the function evaluation, the semantics of `fail` are those of an uncaught *exception*. An introduction to exceptions in Standard ML can be found in [22]. In Fprolog, the exception can be considered to be caught at the point immediately preceding the unification of the term with the corresponding argument of the relation, where it causes that relation to fail.

The general support for exceptions would be consistent with the rest of the functional support in Fprolog as

1. Function evaluation in Fprolog is innermost nested term first (referred to as *eager*), so the evaluation of the expression term to be raised as an exception can occur **before** the exception is raised and the **value** of the expression returned as the exception value. A lazy functional language with *call-by-need* argument evaluation semantics would require special treatment of the expressions given to the `throw` function.

2. Fprolog permits partially defined functions (where some legal actual argument patterns have no matching left-hand-side in the function definition) and function failure. A more general support for exceptions can be provided for which these are special cases.

If, as in Standard ML [17], a general support for exceptions were provided though the use of `raise` and `handle` operators, then the use of `fail` within Fprolog could be shown to be equivalent to the limited use of those exceptions:

`fail` in Fprolog	$\equiv$	`raise Fail` with declared `exception Fail`
With relation R, argument expressions $e_1, ..., e_n$, and goal $R(e_1, \ldots, e_n)$	$\equiv$	e_i `handle Fail` $\Rightarrow$ ensure *failure* of R at each argument $e_i, i = 1 \ldots n$

7.2.1 A proposal for more general exception support in Fprolog

Standard Prolog [10] has support for exceptions at the level of relations with the predefined `catch` and `throw` meta-logical operators. An exception is generally referred to as a `Ball`.

The format for the use of `throw` is:

```
throw(Ball)
```

where `Ball` is any Prolog term to be propagated as an exception. Similarly the format for the use of `catch` is:

```
catch(Goal,Ball,Handler)
```

where:

`Goal` is a Prolog relational goal potentially containing `throw` subgoals

`Ball` is a term to be unified with the actual argument of any `throw` operators encountered during execution of `Goal`

`Handler` is a subgoal to be called when an exception is caught, i.e. successfully unified with `Ball`

Often, `Ball` and `Handler` will contain common free variables, as a means of propagating values from the `throw`.

The goal:

```
catch(throw(foo),X,write(X))
```

will have the effect of writing "foo" to the output, with the execution proceeding as follows:

1. `catch` calls the subgoal given as its first argument, namely `throw(foo)`.
2. The subgoal `throw(foo)` throws (raises) the ball (exception) `foo`.
3. The ball `foo` propagates to the level of the surrounding `catch` where it is unified with the second argument of the `catch` relation (`X`). If this unification had failed, then the ball continues to propagate to any higher enclosing `catch` relation.
4. With the successful unification of `foo` with `X`, the subgoal `write(X)` given as the third argument to `catch` is called.
5. `foo` is written to the output.

In the context of standard Prolog's `catch` and `throw`, the use of `fail` within defined functions in Fprolog can be treated as:

`fail` in Fprolog	$\equiv$	`if throw(fail) then _ else _`
A goal containing relation R, as in $\ldots, R, \ldots$	$\equiv$	$\ldots$,`catch(`R`,fail,fail)`,$\ldots$

Note the use of `if-then-else` to map the relational call to `throw` into an expression. The implicit `catch` which can be considered to be wrapped around each relation call containing functional arguments is shown to only handle one value of exception (`fail`). The `catch` goal will then fail if this type of exception is caught.

With this definition we arrive at the semantics for our use of `fail` within functions as uncaught exceptions, leading to failure of the associated relation.

The definition using `catch` and `throw` could lead to the more flexible use of exceptions within the functional component of Fprolog, although the implementation to date only permits the support for `fail`.

An improved support would:

- Allow any term to be raised as an exception value within a defined function, for example `throw(foo)`.
- Allow exceptions to be caught within the functions rather than propagating to the relational level.

- Treat any uncaught exception from a functional evaluation as `fail`.

The implementation would require the following:

- The meta-relation `throw` should be mapped to a similar function `throw/1`, where an expression `throw(X)` would have the same meaning as
  `if throw(exception(X)) then _ else _`.
  The definition would use the support in Fprolog for relations as if-conditions. The use of anonymous variables as the alternate expressions is arbitrary, as the function `throw` would never return any value. Function could then use `throw` within any expression.

- As with the meta-relation `catch`, a functional equivalent would allow the handling of exceptions at any level of a nested functional expression, as in Standard ML. The ML syntax is
  E `handle` $P_1 => E_1|\ldots|P_n => E_n$
  where E is the expression which may possibly raise an exception, P_i is an expression matching the exception and E_i is the corresponding value to be returned instead of E. The equivalent support in Fprolog would be by nested applications of a `catch` function, which would have the same capabilities as `catch(`E`,exception(`P_i`),`E_i`)` for each pattern P_i for unification with the exception term thrown.

- The implicit `catch` wrapper around each relation R would be
  `catch(`R`,exception(X),fail)`.
  This can be contrasted with the more limited form supporting `fail` given above.

8 Unit

ML has a built-in type '`unit`' with only one member, namely "`()`". A function of intended arity zero will be defined of type "`unit` $\rightarrow \alpha$", and the value of that function will be returned by the explicit application of that function to "`()`".

An example ML function definition of this type is:

```
>fun foo () = 22;
foo: unit -> int
>val a = foo ();
a = 22 : int
```

In Fprolog, all functions are explicitly applied to a **list** of actual arguments, using Prolog syntax for lists, and the application of a function to **no** arguments can be explicit by using an empty list (i.e. nil: "`[]`"). The application of a function to no arguments simply returns that function, i.e.
`foo @ []` for defined function `foo` with arity $0 \equiv$ evaluated `foo`
`bah @ []` for arity `bah` > 0 is $\equiv$ `bah`
$\Rightarrow$ `bah @ [] @ [] @ []` $\equiv$ `bah`
$\Rightarrow$ `bah @ [] @ [] @ [X]` $\equiv$ `bah @ [X]`.

9 The interaction of functions and relations

In the combined functional and logic programming paradigm of Fprolog, most effort has been placed in the design of the overlap between the use of defined functions and relational rules. The resultant system allows the exploitation of defined functions within rules and access to relations from within functions in a straightforward way with clear semantics.

The interaction between the functional and logic elements of a Fprolog program is limited to:

Function definition. The relation `fun` is given special meaning as declaring the ordered equational rewrite rules defining a named function.

Function application. The semantics of the actual argument terms of predicates has been extended to include the application of defined functions with the special operator `@`. The functional reduction is defined to occur as a step preceding the unification of the resultant term with the predicate formal arguments.

Function failure. Function failure is defined, such that a goal with a failing function as an argument term is defined to fail.

Relation call from within functions. The condition term of the built-in Fprolog function `if` is defined to be a relational goal, with determinism ensured by one-solution call semantics.

Functions as goals. The non-curried application of a defined function as a goal is defined to be equivalent to the `=` goal with that application term and `true`.

Functions as first-class data items. A function abstraction returned as the result of a higher-order function or the user definition of a lambda-term can be unified with a logical variable for application within subsequent goals or sub-goals.

10 Some Fprolog examples

A comprehensive review of the application of Fprolog to both logic and functional problems is given in Chapter (Case Studies Chapter Ref).

Fprolog examples of functions for factorial, append, map, and max have been given in the preceding sections, and are repeated here for clarity:

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

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

fun map(F,[]) = [];
    map(F,[X|Xs]) = [F @ [X]|map(F,Xs)].

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

10.1 Undergraduate Prolog exercise attempt

An interesting example of functional logic syntax could be seen in an attempt by an undergraduate to write a relation `remhigh/2` in which the first argument is a list of integers, and the second is the same list excluding the highest element. The undergraduate wrote:

```
%%%% remhigh:    L2 is list L1 excluding highest member of L1

remhigh(L1,L2) :- remove( max(L1), L1, L2).

%%%% remove(Element, List, Remainder_list) :
%%%%     Remainder_list is List excluding Element

remove(N, [N|T], T).
remove(N, [H|T], [H|T1]) :- N \== H, remove(N, T, T1).
```

From the definition of `remhigh` it can be seen that the student expected a functional support that is not present in Prolog. The student is also suggesting a natural syntax. The above attempt would be correct in Fprolog with the definition of `max` given above in Section 5.

10.2 Lazy lists

This example is extended and reviewed in more detail in Chapter (Case Studies Chapter Ref), where infinite streams of primes are created. Here we will show the use of the higher-order features of Fprolog to represent infinite lists.

Infinite lists in this program will be represented by constructor terms of the form:

`item(Head,Tail)`

where `Head` is the value at the head of the list and `Tail` is a *function* of arity zero which returns the tail of the list. The empty list can be represented by a constructor such as `empty`. The functions to extract the components of a list are:

```
fun head(empty)     = fail;
    head(item(X,_)) = X.

fun tail(empty)     = fail;
    tail(item(_,F)) = F@[].
```

A function to create the infinite list of natural numbers is:

```
fun make_nats(N) = item(N,lambda([],make_nats(N+1))).
```

The application `make_nats(N)` can now be used to represent an infinite list the natural numbers starting from N.

A goal such as `?- Z = head(tail(tail(make_nats(1))))`. will return the expected solution `Z = 3`. With this representation of infinite lists, a version of the higher-order function `map` can be defined in Fprolog:

```
fun imap(F,empty)     = empty;
    imap(F,item(X,T)) = item(F@[X], lambda([],imap(F,T@[]))).
```

The function can be demonstrated in a goal such as

```
?- Z = head(tail(tail(imap(*2,make_nats(1))))).
```

giving the solution `Z = 6`.

The `imap` function illustrates the combined use of *constructors* (`empty`,`item`), *higher-order variables* (`F`), explicit application with `@`, implicit application of `imap`, use of `lambda` expressions, and the use of *nil* to denote evaluation of an arity/0 function. The example shows that the syntax facilitates the use of these capabilities without obscure programming constructs.

11 Comparison of Fprolog with `call/N`, `apply/3`

The semantics of the support for functions in Fprolog has most in common with Naish's `apply/3` [20], although he retains the definition of functions as Prolog relations, and permits non-deterministic evaluation. Naish's definition of `apply/3` is designed as a more capable replacement for the `call/N` extra-logical predicate provided in some Prologs and used as the basis for the higher-order functional support in Mercury [26].

Table 4 compares Fprolog with `call/N` and `apply/3` using the examples from [20].

<table>
<tr><th>call/N</th><th>apply/3</th><th>Fprolog</th></tr>
<tr><td><pre>
map(F,[],[]).
map(F,[X|Xs],[Y|Ys]) :-

    call(F,X,Y),
    map(F,Xs,Ys).
</pre></td><td><pre>
map(F,[],[]).
map(F,[X|Xs],[Y|Ys]) :-

    apply(F,X,Y),
    map(F,Xs,Ys).
</pre></td><td><pre>
fun map(F,[])   = [];
    map(F,[X|Xs]) =
        [F @ [X]|map(F,Xs)]
</pre></td></tr>
<tr><td><pre>
filter(P,[],[]).
filter(P,[X|Xs],Ys) :-
    (call(P,X) ->
       Ys = [X|Z]
    ;
       Ys = Z
    ),
    filter(P,Xs,Z).
</pre></td><td><pre>
filter(P,[],[]).
filter(P,[X|Xs],Ys) :-
    (apply(P,X,true) ->
       Ys = [X|Z]
    ;
       Ys = Z
)
    filter(P,Xs,Z).
</pre></td><td><pre>
fun filter(P,[])   = [];
    filter(P,[X|Xs]) =
        if (P @ [X])
        then [X|filter(P,Xs)]
        else filter(P,Xs).
</pre></td></tr>
<tr><td><pre>
foldr(F,B,[],B).
foldr(F,B,[X|Xs],R) :-
    foldr(F,B,Xs,R1),
    call(F,A,R1,R).
</pre></td><td><pre>
foldr(F,B,[],B).
foldr(F,B,[X|Xs],R) :-
    foldr(F,B,Xs,R1),
    apply(F,X,FA),
    apply(FA,R1,R).
</pre></td><td><pre>
fun foldr(F,B,[])   = B;
    foldr(F,B,[X|Xs]) =
        F @ [X,foldr(F,B,Xs)].
</pre></td></tr>
<tr><td><pre>
compose(F,G,X,FGX) :-

    call(G,X,GX),
    call(F,GX,FGX).
</pre></td><td><pre>
compose(F,G,X,FGX) :-

    apply(G,X,GX),
    apply(F,GX,FGX).
</pre></td><td><pre>
fun compose(F,G,X) =
        F @ [G @ [X]].
</pre></td></tr>
<tr><td><pre>
converse(F,X,Y,FYX) :-

    call(F,Y,X,FYX).
</pre></td><td><pre>
converse(F,X,Y,FYX) :-

    apply(F,Y,FY),
    apply(FY,X,FYX).
</pre></td><td><pre>
fun converse(F,X,Y) =
        F @ [Y,X].
</pre></td></tr>
</table>

Table 4: Comparison of `call/N`, `apply/3` and Fprolog

The above relations and functions are then tested against the queries in Table 11 [20].

With the syntax shown in the right-hand column, Fprolog can support the

	call/N, apply/3	Fprolog
1.	filter(>(5),[3,4,5,6,7],As)	As = filter(>(5),[3,4,5,6,7])
2.	map(plus(1),[2,3,4],As)	As = map(+1,[2,3,4])
3.	map(between(1),[2,3],As)	⇒ non-deterministic function
4.	map(plus(1),As,[3,4,5])	⇒ reversible map, plus
5.	map(plus(X),[2,3,4],[3,4,5])	⇒ reversible plus
6.	map(plus(X),[2,A,4],[3,4,B])	⇒ reversible plus
7.	map(plus(X),[A,3,4],[3,4,B])	⇒ reversible plus
8.	foldr(append,[],[[2],[3,4],[5]],As)	As = foldr(append,[],[[2],[3,4],[5]])
9.	foldr(converse(append), [], [[2],[3,4],[5]], As).	As = foldr(converse(append), [], [[2],[3,4],[5]]).
10.	compose(map(plus(1)), foldr(append,[]), [[2],[3,4],[5]], As).	As = map(+1) @ [foldr(append,[]) @ [[2],[3,4],[5]]].
11.	foldr(compose(append,map(plus(1))), [], [[2],[3,4],[5]], As).	As = foldr(compose(append, map(+1)), [], [[2],[3,4],[5]]).
12.	map(plus,[2,3,4],As).	As = map(+,[2,3,4]).

Table 5: Queries from [20] for `call/N`, `apply/3`, Fprolog

functional examples given in [20] with the exception of those requiring multiple answers (3) or reversible functions (4-7). `Call/N` does not provide reversible functions (4-7) or permit general higher-order programming as in (11-12). `Apply/3` does not provide reversible functions (4-7). A discussion of the significant features of each example is given below (and in [Nai96]), followed here by some more examples highlighting the capabilities of Fprolog.

1. `filter(>(5),[3,4,5,6,7],As)`
   The function $>$ passed to `filter` is curried, representing the boolean function $\lambda x \rightarrow (5 > x)$. The higher-order function `filter` applies this argument to `[3,4,5,6,7]`, returning `[3,4]`. The example exercises the definition of higher-order functions and currying.

2. `map(plus(1),[2,3,4],As)`
   In a similar fashion to example 1, the curried function `plus(1)` is passed to the higher-order function `map`. In Fprolog the function and predicate name-spaces are distinct (see Section 6), so the plus function can be given the name `+` rather than a special relation being needed. The Fprolog library includes the definitions of all the arithmetic func-

tions, e.g. `fun +(X,Y) = if (Z is X+Y) then Z else fail.`. The `is` relation is redundant in Fprolog.

3. `map(between(1),[2,3],As)`
   The relation `between(I,J,X)` has multiple solutions, and its call from within a functional expression in Fprolog such as
   `if between(1,9,N) then N else 0`
   would ensure deterministic execution of the predicate. This would enforce a single solution or failure. Example 3 has no equivalent in the functional component of Fprolog, as that would conflict with the implementation on the Delphi Machine.

4. `map(plus(1),As,[3,4,5])`
   Examples 4 through 7 require the functions `map` or `plus` to be reversible. None of `call/N`, `apply/3` or Fprolog provides support for reversible functions.

5. `map(plus(X),[2,3,4],[3,4,5])`
   See 4 above.

6. `map(plus(X),[2,A,4],[3,4,B])`
   See 4 above.

7. `map(plus(X),[A,3,4],[3,4,B])`
   See 4 above.

8. `foldr(append,[],[[2],[3,4],[5]],As)`
   The higher-order function `foldr` accepts a function abstraction (in this case the function `append`) and recursively applies it to the argument list, treating the argument `[]` and the final element. With `call/N` and `apply/3`, the first call to `append` is with the last element of the list of lists and `[]`, e.g. `append([5],[],R)`, where `R` is an intermediate result. Similarly, Fprolog stacks the intermediate result of `append([5],[])`. Each call to `append` is with both required arguments ground, and `call/N`, `apply/3` and Fprolog provide the flattened solution `[2,3,4,5]`.

9. `foldr(converse(append),[],[[2],[3,4],[5]],As)`
   The example proceeds in a similar manner to example 8, with the function abstraction provided by `converse(append)`. When called by `foldr`, the abstraction is passed both required arguments which are appended in reverse, resulting in the solution `[5,3,4,2]`.

10. `compose(map(plus(1)),foldr(append,[]),[[2],[3,4],[5]],As)`
    This is a more complex combination of currying and higher-order functions, but with similar system requirements to examples 8 and 9. `map(plus(1))` increments each member of a list, while `foldr(append,[])`

flattens a list of lists, so the term represents:
increment_list(flatten_list([[2],[3,4],[5]])).
This can be represented more naturally in Fprolog than in the flat syntax with `call/N` and `apply/3`.

11. `foldr(compose(append,map(plus(1))),[],[[2],[3,4],[5]],As)`
This example is evaluated successfully with `apply/3` and in Fprolog, but **not** with `call/N`. The composition of `append` and `map(plus(1))` results in a function which increments the elements of an argument list, and returns a function which prepends that result onto its argument (i.e. `compose(append,map(+1)) @ [[1,2,3]]` $\rightsquigarrow \lambda x \rightarrow$ `append([1,2,3],`x`)`). This abstraction can be passed to `foldr` to be recursively applied to the argument list `[[2],[3,4],[5]]` and `[]` producing `[3,4,5,6]`. The problem that `call/N` has with this example stems from the fact that an intermediate result is produced which is a function abstraction. `Call/N` requires that the right number of arguments must be given for the call to work correctly. For example, `call(plus(1),2,Z)` works correctly giving `Z = 3`, but `call(plus,1,X)` results in an error or fails. This limitation of `call/N` provides the motivation for Naish [20] to recommend `apply/3` in which every application is to one argument and a closure is returned if the function is defined with more.

12. `map(plus,[2,3,4],As)`
In this case, the application of `map` must return an array of function abstractions, highlighting the weakness of `call/N` as in example 11. Fprolog and `apply/3` both produce the expected result, which can be tested in a query such as
`?- map(plus,[2,3,4],[Fa,Fb,Fc]), apply(Fb,5,Z).`
or for Fprolog
`?- [Fa,Fb,Fc] = map(plus,[2,3,4]), Z = Fb @ [5].`
giving the solution `Z = 8`.

The examples above illustrate the limitations of `call/N` and show the similarities of `apply/3` and Fprolog for non-deterministic functions. Other examples will highlight the syntactic advantages of Fprolog over `apply/3`, in Table 11.

1. `Apply/3` consistently treats all functions as relations, such that the flat form of arithmetic expressions is retained with the `is` relation, as in the example with the definition of `inc`. The functional support in Fprolog allows direct use of nested arithmetic expressions, so the `is` relation is redundant. In fact, if `is` appears in a Fprolog goal with an arithmetic argument, the argument will be evaluated before unification

	apply/3	Fprolog
1.	`inc(X,Y) :- Y is X+1.`	`fun inc(X) = X+1.`
2.	`fact(1,1).`	`fun fact(1) = 1;`
	`fact(X,Y) :- X \== 1,`	`    fact(N) = N * fact(N-1).`
	`             X1 is X-1,`	
	`             fact(X1,Y1),`	
	`             Y is X * Y1.`	
3.	`apply4(F,A1,A2,R) :-`	`F = plus, Z = F @ [1,2].`
	`    apply(F,A1,F1),`	
	`    apply(F1,A2,R).`	
	`F = plus, apply4(plus,1,2,Z).`	
4.	`divby2(X,Y) :- Y is X / 2.`	`map(lambda([X],X/2),[2,4,6]).`
	`map(div_by_2,[2,4,6]).`	
5.		`fun div_by_n(N) = lambda([X],X/N).`
		`Z = div_by_n(2) @ [10].`
6.	`fib(0,0).`	`fun fib(0) = 0;`
	`fib(1,1).`	`    fib(1) = 1;`
	`fib(N,M) :- N > 1,`	`    fib(N) = fib(N-2) + fib(N-1).`
	`            N2 is N-2,`	
	`            fib(N2,M2),`	
	`            N1 is N-1,`	
	`            fib(N1,M1),`	
	`            M is M2 + M1.`	
7.	`ffib(F,0,M) :- apply(F,0,M).`	`fun ffib(F,0) = F @ [0].`
	`ffib(F,1,M) :- apply(F,1,M).`	`    ffib(F,1) = F @ [1].`
	`ffib(F,N,M) :- N > 1,`	`    ffib(F,N) =`
	`               N2 is N-2,`	`        F @ [ffib(F,N-2) + ffib(F,N-1)].`
	`               ffib(F,N2,M2),`	
	`               N1 is N-1,`	
	`               ffib(F,N1,M1),`	
	`               MM is M2 + M1,`	
	`               apply(F,MM,M).`	

Table 6: Further programming examples showing Fprolog capabilities

with the corresponding `is` formal parameter. This means that `Z is 1 + 2` $\equiv$ `R = 1 + 2, Z is R`. `is` has quite asymmetric functionality in which the first argument must be a number or a variable while the second argument can also be an arithmetic expression. `(1 + 2) is Z` is not permitted in standard Prolog both for `Z` a variable or with `Z` instantiated to a number. In Fprolog the use of = with library functions provides more consistent support for arithmetic, allowing both `Z = 1 + 2` and `(1 + 2) = Z`. The bracketed terms are for clarity, and `1 + 2 = Z` is equally acceptable.

2. The example of the factorial function `fact` shows that deterministic functions in the relational style must have guard conditions in subsequent clauses (i.e. `X \== 1`) to prevent erroneous non-deterministic execution. For more complex functions the conditions can obscure

the meaning of the code, and Prolog's *cut* is used to provide an efficient solution. `apply/3` does not attempt to address the presence on `cut` to ensure determinism in functions, while Fprolog has consistent deterministic functional evaluation.

3. The use of `apply/3` provides consistent support for higher-order functional programming, but suffers from the implicit treatment of all function applications as nested applications to one argument and the flat representation of application terms. The example shows the application of an arity/2 function to two arguments, and Naish [20] suggests the definition of an auxiliary relation `apply4` to mitigate this difficulty. Fprolog allows the application of functions to an arbitrary number of arguments in a single term.

4. Without nameless functions, the use of `apply/3` requires that defined functions are created for each requirement, and the chosen name used in the place of the lambda expression in Fprolog. The example shows the specification of a function which divides-by-two. The issue with `apply/3` is mitigated by the use of currying, such that if the required function were times-by-two, then a curried application, for example `times(2)`, could by used. In general, however, an auxiliary fact will be needed, as the example shows.

5. The use of defined functions as an alternative to lambda-expressions with `apply/3` is unsatisfactory where the lambda-expression contains free logical variables. The example shows such an expression in the definition of `div_by_n`, and the issue would similarly arise within a goal such as `?- N = @, Z = lambda([X],X/Z) @ [10].` The implementation with `apply/3` would require the use of the Prolog extra-logical relation `assert` or the accumulation of free variables as additional arguments to the auxiliary functions.

6. The eager argument evaluation semantics of Fprolog is equivalent to the flattened form of Prolog relations used with `apply/3`. The example of the Fibonacci function shows the syntax of Fprolog to be a better match to the requirement.

7. The awkwardness of the flattened form with `apply/3` is exacerbated when nested expressions and higher-order applications appear in the function definition. The example gives a modified Fibonacci function where an additional parameter specifies a function (`F`) to be applied to the sub-terms before summation in the recursive case.

   The `apply/3` example of `ffib` illustrates the following differences with Fprolog:

(a) The condition `N>1` is required as `apply/3` has no special consideration for deterministic execution, and assumes use of additional conditions or *cut*.

(b) All expressions with `apply/3` retain their flat Prolog form, leading to an unwieldy syntax for expressions which would naturally be nested.

(c) Arithmetic with `apply/3` relies upon the use of the special Prolog relation `is`. In Fprolog arithmetic expressions can appear anywhere as a valid argument term, and will be reduced before the term is unified with the corresponding formal argument.

(d) Function application in Fprolog can be either explicit with the `@` operator, or implicit by using a defined function name in a compound argument term. The latter case is defined to be syntactic sugaring for the former. The definition of `ffib` using `apply/3` differentiates between the application of a higher-order term represented by the variable `F` in `apply(F,MM,M)` and the recursive call to the function `ffib` in `ffib(F,N2,M2)`. For consistent use of `apply/3`, the recursive call would be replaced by `apply(ffib(F,N2),M2)` which would be converted by `apply/3` to the call `ffib(F,N2,M2)`. It is unclear whether it is better to make consistent use of `apply/3` in higher-order functions and render the non-curried calls more obscure, or whether a mix of `apply/3` and normal relation calls should be used.

12 Conclusions

Higher-order functions can be neatly integrated with Prolog's relations with a deterministic evaluation semantics compatible with the requirements of a Delphi implementation.

Examples given in this and the following two chapters show the capabilities chosen for implementation in Fprolog to be sufficient to express a wide range of programs without resorting to artificial or obscure coding devices.

The capabilities of Fprolog, including the definition and application of functions, the call-once semantics of the `if` condition, and the use of boolean functions as relations, have proved sufficient to preclude the need for *cut* in all the test programs reviewed to date.

13 Summary

The functional component of Fprolog extends the Prolog language in the following ways:

- The definition of functions through the `fun` relation
- The application of functions through the `@` operator
- Support for higher-order functional programming through the use of lambda-expressions and currying
- A general strict functional evaluation semantics with the single exception of a pre-defined `if` function
- Use of relations within functions is limited to the condition argument of the `if` function, where deterministic search for the first solution is enforced
- Support for boolean functions to be treated as relations

These features have proved consistent in use and sufficient to implement a wide range of sample programs without resorting to *cut*.

The defined semantics permit an efficient implementation on an extended Delphi Machine, where function applications embedded within a Prolog program are compiled to direct machine-code calls. Such an implementation has been produced in Fprolog.

References

[1] K. R. Apt, J. W. de Bakker, and J. J. M. M. Rutten, editors. *Logic Programming Languages: Constraints, Functions, and Objects.* MIT Press, 1993.

[2] S. Bonnier. A formal basis for Horn Clause logic with external polymorphic functions. Technical Report 276, Dept. of Computer Science, Linköping University, Sweden, 1992.

[3] S. Bonnier and J. Maluszyńsk. Towards a clean amalgamation of logic programs with external procedures. In R. Kowalski and K. A. Bowen, editors, *Proceedings of the 5th Intl. Conf. and Symposium on Logic Programming*, volume 1, pages 311–326. MIT Press, 1988. re. S-Unification.

[4] M. M. T. Chakravarty and H. C. R. Lock. The implementation of lazy narrowing. In *Proc. 3rd Intl. Symposium Programming Language Implementation and Logic Programming*, pages 123–134. Springer-Verlag, 1991.

[5] P. H. Cheong and L. Fribourg. Implementation of narrowing: The Prolog-based approach. In Apt et al. [1].

[6] W. F. Clocksin. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[7] W. F. Clocksin. *Clause and Effect, Prolog for the Working Programmer.* Springer-Verlag, 1997.

[8] W. F. Clocksin and H. Alshawi. A method of efficiently executing Horn Clause using multiple processors. Technical Report CCSC-3, SRI International (Cambridge Computer Science Centre), 1987.

[9] H. B. Curry. Grundlagen der kombinatorischen Logik. *American Journal of Mathematics*, 52:509–536, 789–834, 1930.

[10] P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer, 1996.

[11] M. Hanus. The integration of functions into logic programming: from theory to practice. *Journal of Logic Programming*, 19,20:583–628, 1994.

[12] M. Hanus, S. Antoy, H. Kucken, and F. López-Fraguas. Curry, an integrated functional logic language. Technical report, RWTH Aachen, Germany, 1997.

[13] F. Henderon, T. Conway, Z. Somogyi, and P. Ross. *The Mercury Language Reference Manual.* University of Melbourne, 1995.

[14] C. S. Klein. *Exploiting OR-Parallelism in Prolog using Multiple Sequential Machines.* PhD thesis, Computer Laboratory, Cambridge University, England, February 1991. Reprinted as Technical Report No. 216.

[15] J. W. Lloyd. Combining functional and logic programming languages. In *Proc. 1994 Intl. Logic Programming Symposium*, 1994.

[16] J. Maluszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. In Apt et al. [1]. re. S-Unification.

[17] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML.* MIT Press, 1990.

[18] J. J. Moreno-Navarro and M. Rodriguez-Artalejo. Logic programming with functions and predicates: The language Babel. *Journal of Logic Programming*, 12:191–223, 1992.

[19] L. Naish. Adding equations to NU-Prolog. In J. Maluszyński and M. Wirsing, editors, *Proceedings of the 3rd Intl. Symposium Programming Language Implementation and Logic Programming*, pages 15–26. Springer-Verlag, August 1991.

[20] L. Naish. Higher-order logic programming in Prolog. Technical Report 96/2, Dept. of Computer Science, University of Melbourne, Australia, 1996.

[21] L. C. Paulson. ML Exercise Sheets, Part 1A CST and Mathematics with Computer Science. Technical report, Computer Laboratory, Cambridge University, England, 1988.

[22] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.

[23] Lawrence C. Paulson and Andrew W. Smith. Logic programming, functional programming, and inductive definitions. In P. Schroeder-Heister, editor, *Extensions of Logic Programming*, LNAI 475, pages 283–310. Springer, 1991.

[24] S. Saraswat. *Performance Evaluation of the Delphi Machine*. PhD thesis, Computer Laboratory, Cambridge University, England, December 1995. Reprinted as Technical Report No. 385.

[25] M. Schönfinkel. Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, 92:305–316, 1924.

[26] Z. Somogyi, F. J. Henderson, and T. C. Conway. Mercury, an efficient purely declarative logic programming language. Technical report, Dept. of Computer Science, University of Melbourne, Australia, 1995.

[27] M. Stickel. A Prolog technology theorem prover: Implementation by an extended Prolog compiler. *J. Auto. Reas.*, 4(4):353–380, 1988.

[28] K. L. Wrench. *A Distributed AND-OR Parallel Prolog Network*. PhD thesis, Computer Laboratory, University of Cambridge, December 1990. Available in summary form as Technical Report No. 212.