On Warren's Method for Functional Programming in Logic

M.H.M. Cheng[*], M.H. van Emden[*], and B.E. Richards[*]
University of Victoria

October 3, 1989

Abstract

Although Warren's method for the evaluation in Prolog of expressions with higher-order functions appears to have been neglected, it is of great value. Warren's paper needs to be supplemented in two respects. He showed examples of a translation from λ expressions to clauses, but did not present a general method. Here we present a general translation program and prove it correct with respect to the axioms of equality and those of the λ-calculus. Warren's paper only argues in general terms that a structure-sharing Prolog implementation can be expected to efficiently evaluate the result of his translation. We show a comparison of timings between lisp and a structure-copying implementation of Prolog. The result suggests that Warren's method is about as efficient as the Lisp method for the evaluation of λ expressions involving higher-order functions.

1 Introduction

In [9] Warren discussed whether higher-order features are needed in Prolog. We are concerned with the first half of the paper, which considers functions and relations that are higher-order in the sense of Lisp and other languages for functional programming. In this paper, as in Warren's, "higher-order" means that functions can be referred to as objects, a usage that is different from the one exemplified by "higher-order logic".

Warren presents a way of translating λ expressions to clauses and claims that this translation makes λ expressions superfluous because

- they are unlikely to result in more efficient compilation, and
- they are "mere syntactic sugar" ([9, page 447]).

[*]Dept. of Computer Science, University of Victoria, Victoria B.C., Canada V8W 2Y2. E-mail: mcheng@csr.UVic.ca

As for the first point, Warren explains that his translation allows a structure-sharing implementation of Prolog to efficiently represent the closures needed for higher-order functions. His conclusion ([9, page 448]):

> So, for DEC-10 Prolog at least, providing a specific implementation for higher-order objects would not provide huge efficiency gains over the approach I am advocating, and for most programs the overall improvement would probably be negligible.

It would have been instructive to compare the efficiency of Warren's translation of

$$t : t : t : t : \mathit{succ} : 0 \;\text{ where }\; t = \lambda f.\, \lambda x.\, f : (f : x)$$

evaluated in DEC-10 Prolog with the corresponding evaluation in Lisp, which, after all, *does* have "specific implementations for higher-order objects". As access to DEC-10 Prolog is hard to get nowadays, we had to settle for a structure-copying rather than a structure-sharing implementation of Prolog. Warren's argument for the appropriateness of Prolog does not apply to a structure-copying implementation. *Yet Warren's translation is evaluated about as fast as what we believe is the corresponding expression in Lisp.* See Appendix B for timings and other details.

When the relative merits of Lisp and Prolog are discussed, it is often conceded that Prolog is better in several respects, but that its weak point is in handling functions. As a result there is much interest in amalgamating functional and logic programming. We suspect that these discussions and activities have proceeded in ignorance of Warren's paper, or at least of its true implications. The Science Citation Index, admittedly bad in its coverage of logic programming, lists *no* references to the paper during the first five years after its publication; for 1988 two references are listed.

The unwarranted neglect of Warren's paper may be due in part to its dismissal of λ expressions as "mere syntactic sugar". Warren translated

$$t : t : t : t : \mathit{succ} : 0 \;\text{ where }\; t = \lambda f.\, \lambda x.\, f : (f : x)$$

to

```
apply(t,F,t(F)).
apply(t(F),X,R) :- apply(F,X,U),apply(F,U,R).

?-apply(t,t,U),apply(U,t,V),apply(V,t,W),apply(W,succ,X),apply(X,0,Ans)
```

apparently preferring the latter notation.

We suspect that his paper would have made a big impact if Warren had presented the principles behind his translation from λ expressions to clauses and if he had shown that his method evaluates expressions involving higher-order functions as fast as a language with "specific implementations for higher-order objects".

It is our goal to supply what is missing from Warren's paper. We have already pointed out that his method is fast enough to make Prolog a strong candidate for the type of applications that Lisp is used for. It remains to show the principles behind Warren's translation.

The absence of evaluable functions in pure Prolog forces one to write, for example, goals such as `sum(1,2,X), sum(X,3,Y)`. Most agree (including hard-bitten opponents of "mere syntactic sugar") that this is unbearable; hence the alternative `X is 1+2+3`. What we propose is to allow the second argument of `is` to include ISWIM[2] expressions, which are widely known to have a simple translation to λ expressions. The remainder of this paper is devoted to a method of translating λ expressions to clauses that gives the same result as Warren's on the example in his paper.

As equations play a central role in the method, we review these in the next section. Section 3 is devoted to translation from λ expressions to equations. Section 4 introduces a technique using the equations as rewrite rules. Section 5 shows how a program for translating equations to non-equational clauses can work by *proving* the clauses from the equations and equality axioms. We present our conclusions in section 6.

2 Recursion equations

Our earliest source for equations as a computational formalism is S.C. Kleene's 1936 paper "General recursive functions of natural numbers" quoted in [4]. On page 16 this reference gives the following example of a system of recursion equations:

$$\begin{aligned} f(0) &= 0 \\ g(x) &= f(x) + 1 \\ f(x+1) &= g(x) + 1 \end{aligned}$$

with the comment that f is the main symbol[1] and that g is an auxiliary one.

It is clear that the domain of the function computed is limited to expressions of the form `0+1+...+1` with `+` associating to the left. Hence, we would now call `+1` a postfix unary constructor function and prefer to write it in the usual functional notation. This gives

$$\begin{aligned} f(0) &= 0 \\ g(x) &= s(f(x)) \\ f(s(x)) &= s(g(x)) \end{aligned}$$

where we need to be aware of *three* different types of function: constructors, auxiliaries and the main function.

The above equations are in what we call *value-oriented* style. The reason is that the terms denote the values of the functions being defined rather than the functions themselves. Another possibility is the *applicative* style where the functions themselves are denoted by terms. In this style, the above example becomes

$$f : 0 = 0$$

[1] [4] says that it can easily be verified that the equations determine the function $\lambda x.\, 2 * x$.

$$g : x = s(f : x)$$
$$f : s(x) = s(g : x)$$

The first argument of the application operator : is a term denoting a function. This in contrast to the value-oriented version, where all terms denote numbers. Also, here : is the only evaluable function.

Applicative style has a number of advantages. One is that the constructor functions are easier to recognize. Another is that the application operator : is syntactically easily interchangeable with the abstraction operator λ. Finally, as we shall see in a later section, another advantage of applicative form is that it can be easily transformed into a readily executable Prolog program.

3 Lambda expressions and recursion equations

Both λ expressions and recursion equations can be used to define functions. Lisp provides both alternatives, for example

```
(DEF SQUARE (LAMBDA (X) (* X X)))
(DEF SQUARE (X) (* X X))
```

To us, the first alternative looks like

$$\text{square} = \lambda X.\, X * X$$

and the second looks like

$$\text{square} : X = X * X.$$

Comparison of these alternatives suggests that the abstraction and application operators are interchangeable. This is indeed the case: if we apply both sides of $f = \lambda x.\, B$ (where x is the only free variable of B) to x, then the left-hand side becomes $f : x$ and the right-hand side becomes, by β-reduction, B. Hence we conclude $f : x = B$. Conversely, let us assume that $f : x = B$, where x occurs free in B. Applying abstraction to both sides gives $\lambda x.\, (f : x) = \lambda x.\, B$. According to the extensionality axiom (η-reduction of λ-calculus) we have $\lambda x.\, (f : x) = f$. Hence we conclude that $f = \lambda x.\, B$. Indeed, each of $f = \lambda x.\, B$ and $f : x = B$ is derivable from the other.

When the λ expression $\lambda x.\, B$ has no free variable, it denotes a fully specified function. Hence a constant of logic, such as f, is an appropriate name for such an object. If, on the other hand, $\lambda x.\, B$ does have free variables, say, $x_1, \ldots, x_n$, the function denoted depends on the values of these variables. Such a partially specified object must be named in logic by a term with these variables as arguments. Hence, in the context of logic, the free variables of a λ term are logic variables; they have a status that is different from the *bound* variables, which are λ variables. It is only to the latter that the binding rules apply.

According to the scope rules of λ-calculus it is possible for two different λ-variables to have the same name, as in the following example:

$$\mathit{fourtimes} = (\lambda f.\, \lambda x.\, f : (f : x)) : (\lambda f.\, \lambda x.\, f : (f : x))$$

It is always possible to rename the λ-variables in such a way that different λ-variables have different names; this was called "standard form" by A. Church. Without loss of generality we assume that the lambda expressions to be translated to recursion equations are in standard form. This has the advantage that we can represent λ-variables also by logic variables, which are chosen to be distinct from other logic variables. We regard an abstraction as a term with two arguments, the bound variable and the body; the first is a logic variable (which will not give problems for λ-terms in standard form) and the second is a logic term representing the body. Strictly speaking, according to the rules of logic syntax, we have to represent $\lambda x.\, t : x$ as $\lambda(x, t : x)$, but we trust that no confusion will arise when we stick to the traditional λ-calculus notation as in the first alternative. More information on the relation between λ expressions and equations can be found in [1].

As an example, let us show how to eliminate the λ's from the definition of the "twice" function $t = \lambda f.\, \lambda x.\, f : (f : x)$. The constant on the left-hand side is appropriate, as the right-hand side has no free variables. Changing the outermost λ to an application gives $t : F = \lambda x.\, F : (F : x)$, where F is a logic variable. To remove the remaining λ, we have to introduce an appropriate term to name the right-hand side. As there is a free variable F, the term has to have it as argument. Hence we obtain

$$\begin{gathered} t : F = g(F) \\ g(F) = \lambda X.\, F : (F : X). \end{gathered}$$

One further application and β-reduction gives

$$\begin{gathered} t : F = g(F) \\ g(F) : X = F : (F : X), \end{gathered}$$

which completes the translation to λ-free equations. As this procedure removes a λ at every stage, any λ expression can be converted to λ-free equations by repeatedly applying it. With the removal of each λ, a new function symbol needs to be generated. We call these the *introduced* function symbols.

The following chain of equalities effects the reverse translation, where the introduced function symbols are exchanged for λ's:

$$t \;=\; \lambda X.\, t : X \;=\; \lambda X.\, g(X) \;=\; \lambda X.\, \lambda Y.\, g(X) : Y \;=\; \lambda X.\, \lambda Y.\, X : (X : Y)$$

The first step uses extensionality; the second uses an equation; this alternation continues throughout the derivation.

To automate the translation by means of Prolog, we devise an SLD-derivation with the same effect as the above chain of equalities. This can be done with respect to a logic program consisting of the equations and the clause

$$X = \lambda Y.\, Z \;\leftarrow\; X : Y = Z_1,\; Z_1 = Z,$$

which is a consequence of extensionality and transitivity. Notice that the λ-variable in $\lambda Y.$ is treated as a logic variable. Hence the second occurrence of Y names the same variable. For this example we obtain the following SLD-derivation:

$\leftarrow t = Z$

$\qquad\qquad Z$ substituted by $\lambda Y.\, Z_2$

$\leftarrow t : Y = Z_1,\ Z_1 = Z_2$

$\qquad\qquad$ equation $t : Y = g(Y)$ used

$\leftarrow g(Y) = Z_2$

$\qquad\qquad Z_2$ substituted by $\lambda Y_1.\, Z_4$

$\leftarrow g(Y) : Y_1 = Z_3,\ Z_3 = Z_4$

$\qquad\qquad$ equation $g(Y) : Y_1 = Y : (Y : Y_1)$ used

$\leftarrow Y : (Y : Y_1) = Z_4$

$\qquad\qquad$ reflexivity used

$\Box$ with $t = \lambda Y.\, \lambda Y_1.\, Y : (Y : Y_1)$

This could be done by a Prolog program if we could find a way of detecting when the leftmost goal needs to resolve with an equation and when with the combined extensionality and transitivity axiom.

The problem presented by this choice of alternatives can be solved by using a different predicate symbol for equality in these two cases. The semantics of logic does not present two different predicate symbols to denote the same relation, although there is usually no reason for this to occur. In this case there is the reason that we want to be selective in using the clauses contributing to the definition of the equality relation.

As a result of these considerations, we use the "=" sign only in equations and *eq* elsewhere. Hence we get

$$\begin{array}{l} eq(X, \lambda Y.\, Z) \ \leftarrow\ X : Y = Z_1,\ eq(Z_1, Z). \\ eq(X, X). \end{array}$$

instead of the earlier form

$$\begin{array}{l} X = \lambda Y.\, Z \ \leftarrow\ X : Y = Z_1,\ Z_1 = Z. \\ X = X. \end{array}$$

To prevent attempts to translate variables we insert `nonvar(X)`, so that our translation from equations to λ expressions is effected by the Prolog program:

```
eq( X, lambda(Y,Z) ) :- nonvar(X), X:Y=Z1, eq(Z1,Z).
eq( X, X ).
```

The first argument of `eq` must be a term occurring as a function in the left-hand side of an equation. For example, when the equations are added to the above two clauses to give the Prolog program, the query `?-eq(t,Z)` succeeds with `Z = lambda(F,lambda(X,F:(F:X)))`. Notice here that the resulting value of `Z` is a λ-expression in standard form; hence λ-terms of the form `lambda(U,lambda(U,...))` can never be generated because every instance of `lambda(Y,Z)` introduces new logic variables.

4 Rewriting by resolution

We can now translate λ expressions into sets of recursion equations. Before describing a translation from these equations to Warren's format, it is instructive to consider computing directly with these equations. One method is to consider the equations as logic clauses and make use of them as rewrite rules. If the rules are augmented with a set of equality axioms in the form of a logic program, it becomes possible to "rewrite by resolution". By this we mean that the Prolog derivation mechanism can be made to apply the equality axioms in such a way that they reduce an expression to canonical form. This approach is discussed in detail in [7].

Although the presence of equality axioms guarantees that SLD derivations exist that mimic equational rewriting, Prolog control will not find such derivations when the equality axioms are in their usual form. However, [7] shows that it is possible to specify an alternative of the usual equality axioms for which essentially one SLD-derivation exists, so that SLD-resolution can be used to mimic equational rewriting. It was observed that the reduction of expressions follows a regular pattern, in which the following two steps are repeated until the empty clause is derived:

1. If the left-hand side of the leftmost goal is canonical, apply the reflexivity axiom, otherwise apply the transitivity axiom.
2. If possible, make use of an equation on the leftmost goal. If not, use substitutivity.

The "canonical" test in the first step can be avoided if we instead try to apply transitivity first and apply reflexivity only if transitivity does not reduce the expression. One possible set of axioms that performs rewriting according to the steps above is the following:

```
eq1( X, Z ) :-
        eq2( X, Y ),                % transitivity
        eq1( Y, Z ).
eq1( X, X ).                        % reflexivity

eq2( X:Y, Z ) :-                    % substitutivity
        eq1( X, X1 ),
        eq1( Y, Y1 ),
        eq3( X1:Y1, Z ),
eq2( X+Y, Z1 ) :-
        eq1( X, X1 ),
        eq1( Y, Y1 ),
        eq3( X1+Y1, Z1 ),
eq2( X, Y ) :- X = Y.

eq3( X, Y ) :- X = Y.               % rewriting
eq3( X, X ).
```

```
X+Y = Z :- Z is Z+Y.                % harness machine arithmetic
```

Each of `eq1`, `eq2`, and `eq3` denote equality. By choosing one of these names we can ensure that Prolog control uses the right equality axiom or an equation at the right time. The `eq2` relation must contain clauses for all function symbols. To evaluate the expression `twice:succ:0` we can pose the query `?-eq1(twice:succ:0,X)` to Prolog. The time required for this technique to evaluate a benchmark was measured and is reported in Appendix B with the other results.

These axioms perform the desired rewriting, but they are limited by the principle behind the rewriting strategy: any rewrite-based system spends a large part of its time searching for subexpressions to rewrite. It is this limitation that leads us to the second half of our translation.

5 Translating equations to relational form

Since rewriting is hampered by nested subexpressions, an optimization would be to somehow disassemble expressions before computation. As a simple example, imagine trying to evaluate the expression `2*(3+4)`. We are able to see that the subexpression `3+4` must be computed before the multiplication can be performed. Thus we could first have the computer solve the equation `3+4=Z`, and then evaluate `2*Z=Result`. If we were to "flatten" expressions in this manner before evaluation, the rewrite system could be much expedited. It could spend all of its time solving sequences of simple equations instead of searching for subexpressions. But we can take the process one step further: since we know what the simple equations will be, we can compile them into Prolog code that will compute their value when called. The rest of this section is devoted to such a translation.

The relationalization axiom

$$\forall x_1,\ldots,x_n,v \quad f(x_1,\ldots,x_n)=v \;\leftrightarrow\; \mathit{ff}(x_1,\ldots,x_n,v)$$

states that *ff* relates the value of the function *f* to its arguments. Together with axioms of equality it can be used to prove from an equation its relationalized version. For example, we can use the relationalization axiom

$$\forall x,y,z \quad x:y=z \leftrightarrow \mathit{apply}(x,y,z)$$

and equality axioms to prove from $t(F):X=F:(F:X)$ the logic program clause

$$\mathit{apply}(t(F),X,Z) \leftarrow \mathit{apply}(F,X,U),\ \mathit{apply}(F,U,Z).$$

We use the following method for such proofs. Start by translating the left-hand side of the equation to relational form. In this example we translate $t(F):X$ to $\mathit{apply}(t(F),X,Z)$. We now form the following chain of implications:

$$\mathit{apply}(t(F),X,Z) \;\leftarrow\; t(F):X=Z \;\leftarrow\; t(F):X=U,\ U=Z \;\leftarrow\; F:(F:X)=Z$$
$$\leftarrow\; F:X=U,\ F:U=Z \;\leftarrow\; \mathit{apply}(F,X,U),\ \mathit{apply}(F,U,Z).$$

The implications are justified, respectively, by the relationalization axiom, transitivity, the equation, substitutivity, and relationalization again. The chain proves

$$\mathit{apply}(t(F), X, Z) \;\leftarrow\; \mathit{apply}(F, X, U), \mathit{apply}(F, U, Z),$$

which is the desired relationalized form of the equation $t(F) : X = F : (F : X)$.

The next step is to formalize this chain further by making each link in it into a goal statement in the sense of logic programming in such a way that the entire chain becomes an SLD-derivation. This requires us to be precise about the form of the axioms used: they are the input clauses of the SLD-derivation.

$$\begin{array}{ll}
\leftarrow \mathit{apply}(t(F), X, Z) & \\
 & \mathit{apply}(X, Y, Z) \leftarrow X : Y = Z \\
\leftarrow t(F) : X = Z & \\
 & X = Z \leftarrow X = Y,\ Y = Z \\
\leftarrow t(F) : X = Z_1,\ Z_1 = Z & \\
 & t(F) : X = F : (F : X) \\
\leftarrow F : (F : X) = Z & \\
 & X : Y_1 = Z \leftarrow Y_1 = Y_2,\ Y_2 = Z \\
\leftarrow F : X = Z_2,\ F : Z_2 = Z & \\
 & X : Y = Z \leftarrow \mathit{apply}(X, Y, Z) \\
\leftarrow \mathit{apply}(F, X, Z_2),\ F : Z_2 = Z & \\
 & X : Y = Z \leftarrow \mathit{apply}(X, Y, Z) \\
\leftarrow \mathit{apply}(F, X, Z_2),\ \mathit{apply}(F, Z_2, Z). &
\end{array}$$

The derivation is incomplete; it yields a "conditional answer" [8, 5] which is a clause where the initial goal with the cumulative substitution applied to it is the head and the final goal statement as body. The derivation proves that this conditional answer is logically implied by the input clauses of the SLD-derivation.

As far as we know the first logic program to translate equations to relationalized form was in [6]. In this program it is not clear why the resulting clause is justified by the equation and the axioms. The authors invented the program by having it duplicate the known manual operations. The SLD-derivation shown above suggests a better way: have the translation from equation to relationalized clause in the form of a meta-interpreter that constructs an SLD-derivation giving the desired clause as conditional answer. In this way the resulting clause is obtained directly from its *proof.*

This method does not depend on the meta-interpreter itself being a logic program. However, Prolog is a convenient choice of language. A meta-interpreter yielding conditional answers is the following.

```
Concl if Concl  <-
Concl if Cond   <- decompose(Concl, Front, Goal, Back),
                   Head = Goal,
                   clause(Head, Body),
                   concatenate(Body, Back, Back1),
                   concatenate(Front, Back1, Cond1),
                   Cond1 if Cond.
```

The first goal of the body states that the list `Concl` can be decomposed into a list `Front`, followed by `Goal`, a single selected goal, and the list `Back` of remaining goals. The program on which the meta-interpreter acts is encoded in the meta-language by clauses of the form `clause(Head,Body)` where `Body` is a list of terms representing atoms in the object language.

The meta-interpreter is activated by a query of the form `?- <a> if Z` where `<a>` is a term representing the conclusion of the desired conditional answer. The simple version of the interpreter shown above gives many different conditional answers. To get exactly the one we want requires the addition of some conditions and the right definition of `decompose` and `clause`; see appendix.

6 Conclusions

We have shown a generally applicable method for translating λ expressions to clauses not containing the equality predicate and to do the translation in such a way that the control of Prolog can perform the equivalent of the evaluation of the λ expression. Our method gives the same result as Warren's when applied to his example (((t:t):t):succ):0 where t=$\lambda f.\, \lambda x.\, f : (f : x)$. We do not know whether our method is the same as Warren's, as he does not discuss it in his paper.

We first translate λ expressions to equations. We perform this part of the translation by means of a few small Prolog clauses not discussed in the paper. In the second stage we relationalize the equations, resulting in clauses that are efficiently executable by Prolog. We show how our implemented translation process is a proof that the relationalized version is a logical consequence of the equations and of the equality axioms.

A natural way to incorporate our translation into Prolog is as an enhancement of the `is` built-in predicate. As it stands, its second argument must be an arithmetic expression. A natural generalization, one that leaves the rest of Prolog unaffected, is to allow this second argument to be any λ expression, preferably in modified syntactic form such as ISWIM[2].

Results of computations that are functions present problems in existing languages for functional programming. Lisp arbitrarily disallows the usual λ notation in such cases, forcing the user to explicitly construct "closures" with system functions especially provided for this purpose. Scheme is an improvement in that it accepts λ notation. However it does not display the result. In our approach a result that is a function contains function symbols introduced by the translation. These are of course inscrutable to the user. We therefore translate equations back to λ expressions. In section 3 we derive a Prolog program of two small clauses that performs this translation. Without it, the value of $t : t$ would be given as $g(t)$ where $t : F = g(F)$, and $g(F) : X = F : (F : X)$. This program translates such an answer to $\lambda X.\, \lambda Z.\, (\lambda Y.\, X : (X : Y)) : ((\lambda Y_1.\, X : (X : Y_1))) : Z)$, which of course still needs β-reductions to give $\lambda X.\, \lambda Z.\, X : (X : (X : (X : Z)))$.

In Appendix B we give comparisons of times taken by Prolog and by several Lisp processors to run similar programs. One of these examples is almost pure higher-order processing, while the others were run to make a comparison of speed on tasks not involving higher-order functions. Although we are aware of many ways in which these comparisons can be misleading, we believe them to be useful nonetheless. In the following paragraphs we discuss some possible objections.

We have only compared run time, not memory usage. In the case of Lisp and Prolog this omission is not as serious as with conventional languages. Lisp and Prolog require in principle an amount of memory that far exceeds what is available on any real machine. Therefore both require garbage collection. As a result, inferior memory usage gives rise to more frequent garbage collections, hence causes longer run time.

It may well be objected that our benchmark, the "twice" function, is not representative of higher-order functions. However, there is no clear agreement as to what does constitute a typical set of higher-order functions so the best one can do is to find an approximation that is as close as possible to a pure higher-order function. The "twice" benchmark has the advantage that almost all processing goes into producing functions that produce functions, and so on.

It may also be objected that Lisp is not a fair representative of a language for higher-order functions as this type of programming occurs rarely in Lisp practice. As a result, implementers are not likely to attempt to produce efficient code from higher-order functions. True enough. But what about Prolog? The implementers of Prolog did not even know that higher-order functions can be handled at all. Thus Prolog is subject even more strongly to the same handicap, which makes our observations all the more surprising.

We started comparing the run times for the "twice" benchmark and were cautioned that the favorable showing of Prolog might be due to the fact that a fast Prolog implementation does *everything* faster than a slow Lisp implementation. The other comparisons show that this is indeed the case. Suppose we would first have run the programs not involving higher-order functions. Someone skeptical about the merits of Prolog would probably object: "Yes, I see that this Prolog is faster on the basic list processing stuff, but I have yet to see how it does on the 'twice' function. You can't even express that." Warren's work and ours shows that "twice" not only can be expressed as well as in λ-calculus, but that Prolog evaluates it faster by about the same factor as it does the basic list processing stuff.

This work is based on the premiss that higher-order functions are important in programming. We realize that not everyone agrees with that. There certainly is a considerable discrepancy between the point of view of J.McCarthy, the inventor of Lisp, according to which higher-order functions must have been important enough to cause him to select the λ-calculus rather than recursion equations as the formalism on which to base Lisp. The problems McCarthy had with recursive definitions (see [3]) support our belief that he must have been committed to λ-calculus.

The prevailing view of the subsequent developers and users of Lisp is quite different. For example, in a widely used book on Lisp the only reference to the λ-calculus is contained in the following quote ([10], page 121):

> Some Lisp aficionados attach a great deal of significance to the role of lambda in Lisp. This may be because the lambda notation is similar to something called the *lambda calculus*, first developed by the logician Alonzo Church. Having the lambda notation seems to give Lisp an air of intellectual respectability that is generally lacking in most programming languages.
>
> Lambda is nice because it detaches the idea of a function from the idea of a name. This allows us to have "disembodied" functions, as we have seen above. In addition, it makes Lisp rather elegant internally. However, the whole business is probably less of a big deal than many people would like you to believe. Lambda is useful, but most Lisp programs would work just as well if lambda did not exist as a separate abstraction.

Yet higher-order functions are a powerful programming tool, even though they fail to impress the Lisp community. As a result there arose in the seventies and early eighties a second generation of languages for functional programming such as (in alphabetic order) FP, Hope, KRC, Lispkit Lisp, Miranda, ML, SASL, Scheme, and probably others, where higher-order functions find a more hospitable environment. Warren's work and ours shows that Prolog, with a suitable preprocessor, can be counted among the second-generation languages for functional programming.

7 Acknowledgments

We are grateful to Dan Hoffman and Michael Levy for helpful remarks. The Canadian Natural Sciences and Engineering Research Council provided facilities for the research reported here.

A Relationalization meta-interpreter

```
:- op(100,yfx,':').

%    The first argument of condAnswer is an atom representing the
%  conditional answer itself; the second is a list of the conditions.
%  condAnswer is true if a successful derivation exists using the
%  goal selection strategy given in "dec", and clauses of the
%  form "clause([Head|Body])".  Control over the general form of the
%  derivation is imposed by the forced application of transitivity
%  and the defining equation in condAnswer, and of substitutivity
%  in condAnswer2.

condAnswer(Conc,Cond) :-
        clause([Conc,X=Y]),          % un-relationalize
        clause([eq(Conc,X)]),        % use defining equation,
        condAnswer2([X=Y],Cond).     % and transitivity


%     condAnswer2 is true if a successful derivation exists using the
%  goal selection strategy given in "dec", and the clauses of the
%  form "clause([Head|Body])".

condAnswer2(Conc,Cond) :-
        decompose(Conc,Front,Goal,Back),
        clause([Goal|Body]),
        concatenate(Body,Back,Tail),
        concatenate(Front,Tail,Conc2),
        condAnswer2(Conc2,Cond).
condAnswer2(Conc,Conc) :-
        not(decompose(Conc,F,Goal,B)).


%     decompose(Cond,F,X,B) is true if X is a goal from the list of goals
%  Cond, and concatenate(F,[X|B],Cond).  Here decompose is restricted such
%  that it chooses only goals of the form X=Y.

decompose(Cond,Front,X=Y,Back) :-
        lDiff(Cond,Front,X=Y,Back).


%     lDiff(Xs,F,E,B) is true if concatenate(F,[E|B],Xs).

lDiff([X|Xs],[],X,Xs).
lDiff([X|Xs],[X|Ys],U,Back) :- lDiff(Xs,U,Ys,Back).


%     An expression X is constructed if it is not a variable and
%  it's not atomic.

constructed(X) :- nonvar(X), not(atomic(X)).


clause([ eq( t(F):X, F:(F:X) ) ]).      % Defining equation

%     The following clauses define substitutivity.  They will only
%  be applied if the head expression contains a constructed subexpression.
%  Clauses of this form are required for each constructor.
```

```
clause([X:Y=Z,Y=Y1,X:Y1=Z]) :- constructed(Y).
clause([X:Y=Z,X=X1,X1:Y=Z]) :- constructed(X).
clause([X+Y=Z,Y=Y1,X+Y1=Z]) :- constructed(Y).
clause([X+Y=Z,X=X1,X1+Y=Z]) :- constructed(X).

%     To define the relationship between predicates and equations.

clause([X:Y=Z,apply(X,Y,Z)]).
clause([apply(X,Y,Z),X:Y=Z]).
clause([X+Y=Z,plus(X,Y,Z)]).
clause([plus(X,Y,Z),X+Y=Z]).
```

B Benchmarks

The speeds of Prolog and Lisp were compared on three benchmarks. The first is as pure higher-order processing as we can think of. The other two involve no higher-order functions and emphasize list processing.

The second benchmark suggests that the ratio between the basic speed of the fastest Lisp and Prolog is close to 3.5. Adjusted with this factor, our method of handling higher-order functions is about .7 times as fast as that of this Lisp implementation. The other benchmarks of course give different numbers. Although it does not make sense to average them, the reader may look at them to get an idea of the range of variation.

The names and versions of the software we used are the following:

IBUKI Common Lisp, Release 01/01.

Franz Lisp, Opus 38.91.

ALS-Prolog Version 1.01.

B.1 Technical foreword

All tests were executed on a Sun 3/280s. Participants included ALS Prolog, Franz Lisp, and IBUKI Common Lisp. The resolution of the timer in all languages was $1/60^{\text{th}}$ of a second. Since some of the tests we wished to measure took only a few tenths of a second it was necessary to repeat each benchmark a number of times and measure the total elapsed time. The list-processing tests were each repeated 50 times. For fewer repetitions, garbage-collection often influenced the results and gave poor repeatability. Since the "twice" benchmark took substantially longer, figures for it are the average of only 5 runs.

That our Lisps were of different dialects turned out to be only a minor factor. The largest difference had to do with creating the closures necessary for implementing the "twice" benchmark. The list-processing tests, in fact, are identical except for the calls for reading the internal clock. Thus only one Lisp program will be presented for each of the last two tests.

B.2 The “twice” benchmark

B.2.1 Franz Lisp code

```
(declare (special t0 f))

(defun tw (f) (fclosure '(f) '(lambda (x) (funcall f (funcall f x)))))

(defun succ (x) (+ x 1))

(defun bench ()
     (progn
        (setq t0 (car (ptime)))
        (funcall (funcall (funcall (funcall (tw 'tw) 'tw) 'tw) 'succ) 0)
        (- (car (ptime)) t0)
)   )
```

B.2.2 IBUKI Common Lisp code

```
(defun tw (f) (function (lambda (x) (funcall f (funcall f x)))))

(defun succ (x) (+ x 1))

(defun bench ()
     (progn
        (setq t0 (funcall 'get-internal-run-time))
        (funcall (funcall (funcall (funcall (tw 'tw) 'tw) 'tw) 'succ) 0)
        (- (funcall 'get-internal-run-time) t0)
)   )
```

B.3 The “naïve reverse” benchmark

This is a slight modification of the traditional “naïve reverse” benchmark. Here we allow indefinite nesting of lists within the list to be reversed. Measured time was to reverse the following list:

```
(a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u)))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
 (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
```

```
        (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u))))
        (a b c (d e f (g h) (i)) j k l (m n) (o p) q r (s (t (u)))))
```

B.3.1 Prolog code

```
%     reversed(List,Tsil) is true if Tsil is the reversed version
%  of List.  For example, reversed([a,[b,c],d],[d,[c,b],a]) is true.

reversed([],[]).
reversed(X,X) :- atomic(X).
reversed([X|Xs],Back) :-
        reversed(X,Xb),
        reversed(Xs,Xsb),
        concatenate(Xsb,[Xb],Back).

concatenate([],X,X).
concatenate([X|Xs],Ys,[X|Zs]) :- concatenate(Xs,Ys,Zs).
```

B.3.2 Lisp code

As mentioned above, the code for these tests was created with the goal of having both languages perform the same steps. Thus the Lisp-based version of the reverse benchmark uses `concat` when it would be much more efficient to have used the `list` built-in to concatenate lists. One could argue that Lisp is being handicapped in order to give Prolog a better chance. However, an equivalent optimization to the Prolog code, the use of difference-lists, was also unused.

```
(defun reverse (l)
    (cond
        ((eq l nil) nil)
        ((listp l) (concat (reverse (cdr l)) (cons (reverse (car l)) nil)))
        ( t  l )
)   )


(defun concat (l1 l2)
    (if (eq l1 nil)  l2  (cons (car l1) (concat (cdr l1) l2)))
)
```

B.4 The "permutation" benchmark

In this test, each language was required to generate a list of all permutations of the list `[a,b,c,d,e,f]`. Generating permutations can be done recursively: Assume that you have such a list of the permutations of n elements. To get the list for $n+1$ elements you insert the new element in each possible place in each list.

B.4.1 Prolog code

```
%    permutations(List,Lists) is true if Lists is a list containing
%  all permutations of List.  For example permutations([a,b,c],
%  [[a,b,c],[b,a,c],[b,c,a],[a,c,b],[c,a,b],[c,b,a]]) is true.

permutations([],[[]]).
permutations([X|Xs],Ps) :-
        permutations(Xs,P1s),
        inserted_in_all(X,P1s,Ps).

%    inserted_in_all(Element,Lists,NewLists) is true if NewLists
%  is the list of lists resulting from inserting Element into all
%  possible positions in each of the lists in Lists.  For example,
%  inserted_in_all(a,[[b],[c]],[[a,b],[b,a],[a,c],[c,a]]) is true.

inserted_in_all(E,[],[]).
inserted_in_all(E,[L|Ls],Is) :-
        inserted_in([],E,L,Set1),
        inserted_in_all(E,Ls,Set2),
        concatenate(Set1,Set2,Is).

%    inserted_in(ToTheLeft,Element,List,Lists) is true if Lists is
%  the list of lists resulting from inserting Element into all pos-
%  sible positions in List.  In addition, all lists in Lists have
%  ToTheLeft as a prefix.  For example, inserted_in([a],b,[c,d],
%  [[a,b,c,d],[a,c,b,d],[a,c,d,b]]) is true.

inserted_in(ToLeft,E,[],[List]) :-
        concatenate(ToLeft,[E],List).
inserted_in(ToLeft,E,[X|Xs],[First|Lists]) :-
        concatenate(ToLeft,[X],NewLeft),
        inserted_in(NewLeft,E,Xs,Lists),
        concatenate(ToLeft,[E,X|Xs],First).

concatenate([],X,X).
concatenate([X|Xs],Ys,[X|Zs]) :- concatenate(Xs,Ys,Zs).
```

B.4.2 Lisp code

The Lisp code mirrors the Prolog implementation.

```
(defun permutations (l)
    (if (eq l nil)
        (cons nil nil)
        (insert_in_all (car l) (permutations (cdr l)))
)   )


(defun insert_in_all (e l)
    (if (eq l nil)
        nil
        (concat (all_inserts nil e (car l)) (insert_in_all e (cdr l)))
)   )


(defun all_inserts (to_the_left e l)
    (if (eq l nil)
        (cons (concat to_the_left (cons e nil)) nil)
        (cons
            (concat to_the_left (cons e l))
            (all_inserts (concat to_the_left (cons (car l) nil)) e (cdr l))
)   )   )
```

B.5 Table of results

	IBUKI(I)	Franz(I)	IBUKI(C)	Franz(C)	Prolog(W)	Prolog(=)
twice	67.0	25.6	7.5	19.7	1.5	22.6

	IBUKI(I)	Franz(I)	IBUKI(C)	Franz(C)	Prolog
reverse	1.770	.692	.078	.168	.022
permutation	1.973	.857	.383	.382	.065

In the headings above, (I) means that the Lisp in question was the interpreted version and (C) implies that it was compiled. In the "twice" benchmark, the (=) column represents the rewrite-based system and (W) stands for Warren's format. All timings are in seconds.

References

[1] M.H.M. Cheng. *Lambda-equational Logic Programming.* PhD thesis, University of Waterloo, 1987.

[2] P. Landin. The next 700 programming languages. *Comm. ACM*, 9:157–164, 1966.

[3] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3:184–195, 1960.

[4] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[5] M.H. van Emden. Conditional answers for polymorphic type inference. In K.A. Bowen and R.A. Kowalski, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 590–603. MIT Press, 1988.

[6] M.H. van Emden and T.S.E. Maibaum. Equations compared with clauses for specification of abstract data types. In *Advances in Database Theory*, pages 159–194. Plenum Press, 1981.

[7] M.H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.

[8] P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432, 1986.

[9] D.H.D. Warren. Higher-order extensions to prolog: are they needed? In J.E. Hayes, Donald Michie, and Y-H. Pao, editors, *Machine Intelligence 10*, pages 441–454. Ellis Horwood with John Willey and Sons, 1982. Lecture Notes in Mathematics 125.

[10] Robert Wilensky. *LISPcraft*. W.W. Norton, 1984.