Contributions to Functional Programming in Logic

by

Bradley E. Richards

B.A., Gustavus Adolphus College, Minnesota, 1988

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in the Department
of
Computer Science

[redacted]

We accept this thesis as conforming
to the required standard

[redacted]

Dr. Maarten H. van Emden, Supervisor (Dept. of Computer Science)

[redacted]

Dr. Mantis H.M. Cheng, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. Charles G. Morgan, Outside Member (Dept. of Philosophy)

[redacted]

Dr. Kin F. Li, External Examiner (Dept. of Electrical and Computer Engineering)

ISBN 0-315-62352-7

Supervisor: Dr. Maarten H. van Emden

Abstract

In logic programming, computations are described as relations between inputs and outputs. This is a powerful paradigm, and well-suited to expressing many computations, but there are classes of problems that can be specified more naturally in the functional programming style. Adding functional programming facilities to Prolog results in a more powerful language, as they allow higher-order functional expressions to be evaluated conveniently within the logic programming environment. And, as will be shown in this thesis, the efficiency of functional programming in logic is competitive with other functional evaluation techniques.

This thesis presents two methods for evaluating higher-order functional expressions in logic. The first uses equational function definitions as inputs to a logic-based term-rewriting system. The second compiles the same equations into equivalent clauses that can be directly executed by Prolog. Programs are developed for translating between λ-expressions and equations, for performing rewriting, and for compiling equations into Prolog clauses. In addition, benchmarks are run to compare the evaluation speed of these methods to a pair of Lisp implementations.

Examiners:

[redacted]

Dr. Maarten H. van Emden, Supervisor (Dept. of Computer Science)

[redacted]

Dr. Mantis H.M. Cheng, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. Charles G. Morgan, Outside Member (Dept. of Philosophy)

[redacted]

Dr. Kin F. Li, External Examiner (Dept. of Electrical and Computer Engineering)

Contents

List of Tables

List of Figures

Acknowledgements

This thesis could not have been completed without the guidance of my supervisor, Dr. Maarten van Emden. I thank him for his patience in helping me to understand the many things I have learned, and for giving me as much responsibility as he did during my time at the University of Victoria. In addition, I must apologize to Eric Davies and Craig Sinclair for subjecting them to almost daily doses of an often unripe thesis. Their patience and constructive comments have contributed immeasurably to the quality of the final product. Craig deserves special mention for making sure that I exceeded my recommended daily allowance of caffiene as often as possible, and for letting me get to know Toly. Finally, my thanks to Ann for her constant support.

Chapter 1

Introduction

In logic programming, computations are described as relations between inputs and outputs. This technique works well for many programming tasks, but it forces simple arithmetic expressions to be written in a rather cumbersome form. For example, to add three numbers pure Prolog requires one to use something like the following query:

```
?- sum(1,2,X), sum(X,3,Y).
```

This is in contrast to a language such as Pascal, that allows one to specify the same computation more directly:

```
Y := 1+2+3;
```

The former approach requires more typing, uses an extra variable, and tends to obscure the fact that three numbers are being added.

Early implementors of Prolog recognized these difficulties and allowed simple expressions to be written in a more succinct form:

```
?- Y is 1+2+3.
```

Thus it is easy to evaluate simple arithmetic expressions in Prolog. However, it is still not possible to compute function-valued expressions, or expressions that accept functions as arguments, e.g. the composition of a pair of functions, or the result of applying a function to each element in a list. We see this lack of higher-order functional programming capabilities as a shortcoming, and feel that a mechanism for evaluating such expressions should be added to Prolog.

1.1 Why functional programming in logic?

The functional programming paradigm has proven to be a natural way of expressing many classes of problems, and modern implementations have made much progress towards efficient functional programming systems. Why, then, the interest in adding functional programming capabilities to Prolog when specialized systems already exist? There are three motivations:

convenience: Functional expressions could be evaluated without having to leave the Prolog environment.

research: Implementing functional programming in logic is a step towards a more important goal: the amalgamation of the functional and relational programming styles. This has been an active area of research in the past decade. A number of approaches have been proposed [CY86, GM84, AKN86, Nar88], but efficient implementations have not appeared. If it were possible to evaluate higher-order functional expressions efficiently within Prolog it would make Prolog more attractive as a language in which to implement the amalgamation.

efficiency: The execution speeds of Prolog implementations have rapidly increased in recent years. Any system written in Prolog will continue to take advantage

of advances in Prolog implementation technology. *It may be that functional programming in logic is, or will become, competitive with other functional evaluation techniques.*

1.2 Our solutions

It would certainly be possible to create a Prolog implementation that contained a complete functional programming subsystem in addition to a standard Prolog engine. Whenever a functional expression needed to be reduced, it would be passed to the subsystem for evaluation. While this approach would be relatively straightforward, it suffers from three major disadvantages:

1. It ignores the differences between logic and lambda variables.
2. The size of such a system would be prohibitive. It is not unusual for a Common Lisp package alone to take up many megabytes of both disk space and RAM.
3. It would require us to re-implement Prolog, a task that seems counter-productive given the efficient Prolog implementations at our disposal.

If it were possible to have the Prolog derivation mechanism mimic functional programming, these shortcomings could be avoided since a Prolog implementation alone would be sufficient to perform both logic and functional programming. In addition, the system would be easily portable to future Prolog implementations. Thus our research has focused on a pair of software-based solutions:

Rewriting

Functional programming can be implemented by term rewriting systems if function definitions are expressed as rewrite rules. This is the basis of our first approach. A translation from λ-calculus style function definitions to equivalent sets of equations is shown. The equations are then used as rewrite rules by a logic program that performs "rewriting by resolution".

Compiling

Our second approach borrows the λ-to-equations translation, but uses the sets of equations in a different manner. A technique is described for compiling the equations into directly-executable Prolog code that is a logical consequence of the equations and basic mathematical axioms describing equality. The execution speed achieved by the compiled code rivals that of Lisp implementations.

1.3 Thesis overview

This thesis contains a discussion of previous and related work (Chapter 3), an examination of the translation from λ-expressions to equations (Chapter 4), a description of the rewriting approach in full detail (Chapter 5), and a complete treatment of the compilational approach (Chapter 6). Our conclusions and future directions for research are listed in Chapter 7. But first it is necessary to introduce some background material that the reader may find useful in understanding the rest of this thesis.

Chapter 2

Preliminaries

2.1 The λ-calculus

A complete introduction of functional programming is outside the scope of this thesis. Thus, it will be assumed that the reader has at least a rudimentary knowledge of the paradigm. However, most functional programming implementations are based to some degree on the λ-calculus. This section will provide an overview of λ-calculus and establish the terminology that will be used throughout the thesis.

2.1.1 Introduction

At its most basic, the λ-calculus is a formal system for defining functions and describing how they behave. It provides λ-notation as a syntactic convention for writing functions and function applications. Assuming an infinite sequence of variables (e.g. x, y, f), individual constants (e.g. 1, 2, 3), and function constants[1] (e.g. $+$, $-$,

[1]The word "constants" will be used to mean both individual and function constants throughout this thesis.

$*$), an expression in λ-notation, called a *λ-term*, is defined inductively as one of the following:

an atom: All variables and constants are λ-terms.

an application: If M and N are λ-terms, then $(M\ N)$ is a λ-term.

an abstraction: If M is a λ-term and x is a variable, then $\lambda x.\, M$ is a λ-term.

The λ-notation is motivated by the need to indicate which variables in an expression are meant as formal parameters. For example, the expression $x + y$ can be a function of x, y, or both. Writing functions as λ-abstractions allows one to disambiguate such expressions by stating which variables will be supplied when a function is to be applied. Informally, a term of the form $\lambda x.\, B$ represents a function of x, whose value for an argument N can be determined by substituting N for x in B. But unless care is taken in formalizing the notion of substitution, function application can produce incorrect results. (This will be illustrated in Section 2.1.3.)

2.1.2 λ-variables

Before discussing substitution, it is necessary to distinguish between "free" and "bound" variables in λ-terms. The set of free variables in an expression M, written as $FV(M)$, can be defined inductively as follows:

1. $FV(a) = \{\}$ if a is a constant.
2. $FV(x) = \{x\}$ if x is a variable.
3. $FV(\lambda x.\, M) = FV(M) - \{x\}$.

4. $FV((M\ N)) = FV(M) \cup FV(N)$.

The *scope* of the abstraction operator λx in the term $\lambda x.B$ is B. An occurrence of a variable x is *bound* in a term M if it occurs in a subterm of M of the form $\lambda x.N$, otherwise it is *free*. A *closed* term is one without any free variables.

2.1.3 Substitution

The result of substituting N for every free occurrence of x in M is written as $[N/x]M$, and can be defined as follows:

1. $[N/x]x = N$,
2. $[N/x]a = a$ if $x \neq a$,
3. $[N/x](M_1\ M_2) = ([N/x]M_1\ [N/x]M_2)$,
4. $[N/x]\lambda x.M = \lambda x.M$,
5. $[N/x]\lambda y.M = \lambda y.[N/x]M$ if $x \neq y$ and $y \notin FV(N)$,
6. $[N/x]\lambda y.M = \lambda z.[N/x]([z/y]M)$ if $x \neq y$, $y \in FV(N)$, and $z \notin FV(M\ N)$.

The last rule for substitution is of particular importance, as it prevents substitutions from altering the meaning of an expression[2]. Notice that it performs a renaming of the bound variable y when y occurs free in N. This is to avoid *capturing* the free variable in N. As an illustration, imagine substituting $y + z$ for x throughout the expression $\lambda y.2 * y + x$. Performing the substitution without renaming the bound

[2]Church's original formulation of the λ-calculus prohibited a substitution in case 6, but it has become common to define substitution for all cases at the expense of a more complicated definition. ([HS86] is taken as the authority on λ-calculus.)

variable would result in the expression $\lambda y.\, 2 * y + y + z$. But this is incorrect! The variable y occurred free in $y + z$, and became bound when the substitution was performed. To avoid this, the bound variable must be renamed such that it is distinct from the free variables in both the body and the expression being substituted. This renaming is referred to as *α-conversion.*

2.1.4 β-reduction

It was mentioned earlier that a term $\lambda x.\, B$ represents a function of x, and that it can be evaluated at N by substituting N for x throughout B. Now that substitution has been formalized, this operation can be more precisely described.

Any application of the form $(\lambda x.\, B)N$ is called a *β-redex*, or *redex*, and the corresponding term $[N/x]B$ is called its *contractum.* The result of replacing a β-redex by its contractum is called a β-reduction or β-contraction. An expression M is said to *β-reduce* or *reduce* to M' if M' can be obtained from M by a finite series of β-contractions. An expression containing no redexes is in *normal form.*

A λ-expression can be evaluated by using the β-reduction rule repeatedly until the expression has been reduced to normal form. However, there is often more than one redex awaiting contraction at any given time. The first Church-Rosser theorem guarantees that an expression has at most one normal form. Thus the order in which the redexes are chosen will not influence the result of the reduction. But there are instances in which one reduction strategy reaches the normal form while another may fail to do so. For example, consider the following expression:

$$(\lambda x.\, 3)((\lambda y.\, y\ y)(\lambda y.\, y\ y)).$$

The function $\lambda x.\, 3$ discards its argument, so any time spent evaluating the redex to the right will be wasted. What is worse, attempting to evaluate $(\lambda y.\, y\ y)(\lambda y.\, y\ y)$ first

leads to a non-terminating sequence of reductions! (This makes an interesting exercise. When $\lambda y.\, y\ y$ is applied to $\lambda y.\, y\ y$, both occurrences of y in the body of the first term are replaced by the operand, $\lambda y.\, y\ y$, yielding the expression $(\lambda y.\, y\ y)(\lambda y.\, y\ y)$. Clearly, the β-reduction operation could be applied infinitely many times to this term without reaching a normal form since the expressions before and after a β-contraction are identical.)

Given a redex $(\lambda x.\, M)N$ to reduce, there are two possible courses of action. An attempt can be made to reduce N before performing the contraction, or the reduction can be performed before simplifying the occurrences of N in its contractum. Always choosing to reduce the argument first is referred to as *applicative order*. Giving priority to the contraction is called *normal order*. The second Church-Rosser theorem states that normal order evaluation will always lead to the normal form if one exists.

2.1.5 η-conversion

Intuitively, a pair of functions f and g are equal if $f(x) = g(x)$ for all x. But λ-calculus, as described above, provides no notion of function-equality. The η rule is usually included in a formalization of λ-calculus to express the equivalence of such functions:

$$\lambda x.\, (M\ x) = M \text{ if } x \notin FV(M)$$

For example, the λ-terms y and $\lambda x.\, (y\ x)$ are equivalent by this definition, and it can be seen that the application of either of these terms to an argument z results in the term $(y\ z)$. η-conversion allows any λ-term of the form $\lambda x.\, (M\ x)$ to be replaced by M as long as x does not occur free in M. The term $\lambda x.\, (M\ x)$ is referred to as an η-redex, and M is its contractum.

$$\text{twice} = \lambda f.\, \lambda x.\, f : (f : x)$$

$$\text{succ} = \lambda x.\, x + 1$$

Figure 2.1: Definitions of the "twice" and "succ" functions

2.1.6 Our syntax

The λ-syntax used throughout this thesis differs from that of the pure λ-calculus in several respects:

- In pure λ-calculus, function application is denoted by juxtaposition. This thesis will often use the left-associative infix operator ':' to represent applications.
- Pure λ-calculus uses only prefix notation. In this thesis the use of infix operators such as '+' will be allowed, and it is assumed that the standard operator precedences will disambiguate any expressions in which they occur.

2.2 The "twice" example

The "twice" function (defined in Figure 2.1) is of interest, as it is one of the simplest examples of a higher-order function. It expects a function f as its first argument, and applies f twice to its second argument, x. The function is of value for two reasons. First, it can be used to illustrate how higher-order functions are handled by a given functional programming scheme. Second, it can be used as a benchmark to assess the ability of a functional programming system to evaluate higher-order functions.

$$((\lambda f.\,\lambda x.\,f : (f : x)) \odot (\lambda f.\,\lambda x.\,f : (f : x))) : succ$$
$$(\lambda x.\,(\lambda f.\,\lambda x.\,f : (f : x)) : ((\lambda f.\,\lambda x.\,f : (f : x)) : x)) \odot succ$$
$$(\lambda f.\,\lambda x.\,f : (f : x)) : ((\lambda f.\,\lambda x.\,f : (f : x)) \odot succ)$$
$$(\lambda f.\,\lambda x.\,f : (f : x)) \odot (\lambda x.\,succ : (succ : x))$$
$$\lambda x.\,((\lambda x.\,succ : (succ : x)) : ((\lambda x.\,succ : (succ : x)) \odot x)$$
$$\lambda x.\,(\lambda x.\,succ : (succ : x)) \odot (succ : (succ : x))$$
$$\lambda x.\,(succ : (succ : (succ : (succ : x))))$$

Figure 2.2: A sample reduction involving "twice"

When "twice" is applied to itself and then to another function, it has an exponential effect. That is, n applications of "twice" to a function f results in X_n applications of f where $X_n = 2^{X_{n-1}}$ and $X_1 = 2$. For example, the expression

$$\text{twice} : \text{twice} : \text{twice} : \text{succ} : 0$$

reduces to 16, but four applications of twice evaluates to 2^{16}. A fifth application is impossible on any existing machine. Computing multiple applications of twice requires mainly higher-order function manipulation, and takes long enough in most functional programming implementations for reliable timings to be made. A sample reduction of twice : twice : succ is shown in Figure 2.2. The application to be performed next in each line is marked with a $\odot$.

2.3 Equations as a formalism

Equations can define any computable function. Our earliest source for equations as a computational formalism is S.C. Kleene's 1936 paper "General recursive functions of

natural numbers" quoted in [Rog67]. Kleene was interested in characterizing general recursive functions, and needed a language in which to express them. He devised sets of *recursion equations*, so called because the value of a function for a given argument is defined in terms of the value of the function for an argument that is in some sense simpler, or in terms of a simpler function. On page 16, [Rog67] gives the following example of a system of recursion equations:

$$f(0) = 0$$
$$g(x) = f(x) + 1$$
$$f(x+1) = g(x) + 1$$

with the comment that f is the main function symbol[3] and that g is an auxiliary one.

It is clear that the function f returns expressions of the form $0 + 1 + \ldots + 1$ where the $+$ associates to the left. Apparently, $+1$ should be read as a postfix unary constructor. This is better written in the usual successor notation, giving the following set of equations:

$$f(0) = 0$$
$$g(x) = s(f(x))$$
$$f(s(x)) = s(g(x))$$

There are now *three* different types of function: constructors, auxiliaries, and the main function. In the equations above, terms denote the values of the functions being defined rather than the functions themselves. Hence this format is referred to as *value-oriented.*

The format that will be favored in this thesis is the *applicative* style, where the functions themselves are denoted by terms. In this style, the value of a function is

[3][Rog67] says that it can easily be verified that the equations determine the function $\lambda x.2 * x$.

specified by the application of the function to an argument. So, the equations become:

$$f:0 = 0$$

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

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

Here the first argument of the application operator ':' is a term denoting a function. For example, f here refers to a function, and $f:0$ represents a value. This style provides consistency as ':' 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. (The importance of this interchangeability will become clear in Chapter 4). Finally, as shall be seen in Chapter 6, another advantage of applicative form is that it can be transformed into a readily executable Prolog program.

2.4 Term rewriting systems

Term rewriting systems provide a means of computing with equations of the form $X = Y$. The equations are interpreted directionally, from left to right, as *rewrite rules* which specify that any occurrence of the left-hand side in an expression can be replaced by the right-hand side. The terms on either side of a rewrite rule are allowed to contain variables that become instantiated when the rule is applied. For example, the expression $3 + 0$ could be reduced to 3 by applying the equation $x + 0 = x$.

Much of the terminology from λ-calculus is used in discussing term rewriting systems. A subexpression matching the left-hand side of any rewrite rule is called a

redex, and an expression containing no redexes is in *normal form.* Expressions can be *reduced* by applying rewrite rules repeatedly until the normal form is reached (if one exists). As in λ-calculus, an expression may contain more than one redex. Thus the discussion of reduction orders in Section 2.1.4 applies here as well. In addition, for certain sets of rewrite rules it may be the case that more than one *equation* could be applied to a given redex. This allows the possibility of multiple normal forms, where the normal form reached depends upon the choice of rewrite rules.

It can be shown that the Church-Rosser theorems hold for term rewriting using sets of "well-behaved" equations [HO82], or equations that satisfy the following three conditions:

1. No variable may be repeated on the left-hand side of an equation.
2. If two different left-hand sides match the same expression, then the corresponding right-hand sides must be the same.
3. When two (not necessarily different) left-sides match two different parts of the same expression, the two parts must not overlap.

Chapter 3

Previous work

3.1 Equational programming

In [HO82], Hoffmann and O'Donnell describe the power and utility of equations as a computational formalism. The authors are concerned with the correctness of computations, and feel that programs should be expressed in a language that has a complete proof system so that the results of a computation can be shown to be logical consequences of the program. They are aware of Prolog, but feel that the backtracking strategy employed in the search for answers is too costly for general use. Instead, they propose the language of equational logic, with term rewriting as a computational mechanism. It is shown that if a set of equations satisfies the restrictions in Section 2.4, an expression being reduced in their presence has at most one normal form, and that it can be reached by performing normal-order evaluation.

The main emphasis in the paper is on the creation of an efficient equational programming system. Towards this end, the authors discuss methods for "compiling" equations into data structures that can be efficiently manipulated by an optimized

term rewriting system. They suggest the following applications for an equational programming system:

programming: Programs can be written as sets of equations.

specifying interpreters: The semantics of a language can be described by a set of equations. When these equations are given to an equational programming system, they result in an interpreter that adheres strictly to the semantics of the language. The authors illustrate this by specifying the semantics of both Lisp and a subset of Lucid equationally.

data type implementations: Equations describing data types can be used to automatically produce correct implementations.

theorem proving: When a theorem is of the form $A = B$, it can sometimes be proven by reducing both A and B and arriving at identical expressions.

3.2 Compiling functional languages

3.2.1 Turner's combinators

In theory, it is possible to transform any λ-expression into an equivalent expression built only from applications of primitive functions and *combinators* (or closed λ-expressions). In fact, only two combinators, S and K are required, though a third combinator, I, is usually included for efficiency reasons. (See Figure 3.1.) It was recognized by D.A. Turner that the resulting expressions could be reduced by a machine that implemented the reduction of these three combinators [Tur79]. His language SASL compiled function definitions into combinator form and used such a reduction machine to evaluate them.

$$S = \lambda f.\, \lambda g.\, \lambda x.\, f : x : (g : x)$$

$$K = \lambda x.\, \lambda y.\, x$$

$$I = \lambda x.\, x$$

Figure 3.1: S, K, and I combinator definitions

Compiling into a set of fixed combinators has a number of advantages. The reduction machine is simple, and can be optimized to deal with the required combinators. Turner's machine was also able to implement normal order reduction in times comparable to applicative order. However, there are some serious drawbacks. The translation to combinators is expensive, and the resulting expression is *much* larger than the original. In the worst case, the size of the combinator expression can be proportional to the *square* of the size of the initial λ-expression [Ken82](As cited in [Jon87]).

3.2.2 Hughes' supercombinators

Hughes recognized that the shortcomings of the combinator approach arose from translating λ-expressions into such a small number of fixed combinators. He proposed transforming each function definition into a new combinator, called a *supercombinator* [Hug84]. A supercombinator S is a λ-expression of the form

$$\lambda x_1.\, \lambda x_2 \ldots \lambda x_n.\, E$$

where E is not a λ-abstraction and:

1. S has no free variables,

2. any λ-abstraction in E is a supercombinator,

3. $n \geq 0$.

Any λ-expression can be transformed into a supercombinator through a process known as *lambda lifting* [Joh85], which abstracts the free variables out of the expression. For example, consider the following program:

$$(\lambda x.\underline{(\lambda y.\, x + y)}\ x)\ 4.$$

The inner λ-expression is not a supercombinator because it contains a free variable. However, β-reduction can be used in reverse (called β-abstraction) to produce an equivalent expression that *is* a supercombinator:

$$\underline{(\lambda z.\, \lambda y.\, z + y)\ x}.$$

If the supercombinator $\lambda z.\, \lambda y.\, z + y$ is given a name, say A, the original expression can be rewritten as

$$(\lambda x.\, A\ x\ x)\ 4,$$

which is also now a supercombinator. If the name B is given to $\lambda x.\, A\ x\ x$, the program can be written simply as $B\ 4$, where $B\ x = A\ x\ x$.

Reduction of an expression containing a supercombinator cannot take place until all arguments are present. The arguments are then substituted for the formal parameters simultaneously. Thus supercombinator definitions can be regarded as a set of rewrite rules [Jon87].

3.3 Warren's method

In [War82], D.H.D. Warren examines a pair of possible extensions to Prolog. It had been argued that Prolog lacked the power of functional languages which are able to

treat functions as data objects. These languages allow function applications where the function to be applied is determined at runtime. A similar addition to Prolog would be to permit the predicate symbol in a procedure call to be a variable. Warren gives the following example:

```
have_property([],P).
have_property([X|L],P) :- P(X), have_property(L,P).
```

This program would allow the user to check whether all of the elements of a list satisfy a condition `P`.

Also, most functional languages provide the ability to refer to a functional expression without having to give it a name. This is one of the strengths of the λ-calculus. An analogous extension to Prolog would be to allow the use of λ-notation in the description of predicates, where the body of a λ-expression would be a list of Prolog goals. So, for example, the following query becomes possible:

```
?- have_property( [0,1], lambda(X).square(X,X) ).
```

Which asks if both `0` and `1` have the property of being equal to their squares. (Notice that `X` is a logic variable.)

Warren regards both of these extensions as "merely syntactic sugar", as either can be mapped into existing Prolog. The mapping involves transforming variable procedure calls of the form $P(t_1, \ldots, t_n)$ into the call $\text{apply}(P, t_1, \ldots, t_n)$ and adding clauses such as

$$\text{apply}(\text{foo}, X_1, \ldots, X_n) \leftarrow \text{foo}(X_1, \ldots, X_n)$$

for each predicate 'foo' that needs to be treated as a data object. A nameless predicate $\text{lambda}(X_1, \ldots, X_n).E$ is replaced by some unique identifier ϕ that is then defined by a separate 'apply' clause: $\text{apply}(\phi, X_1, \ldots, X_n) \leftarrow E$.

As a final example, Warren shows a translation of the "twice" example. (See Section 2.2.) He states that with the transformation rules above in conjunction with "the standard technique for translating functional notation into predicate notation", the "twice" example can be evaluated by the following call:

```
?- apply(twice,twice,F1), apply(F1,twice,F2),
                                apply(F2,succ,F3), apply(F3,0,Ans),
```

if the function definitions are first translated into a set of `apply` clauses in Prolog:

```
apply(twice,F,twice(F)).
apply(twice(F),X,Z) :- apply(F,X,Y), apply(F,Y,Z).
apply(succ,X,Y) :- Y is X+1.
```

However, his paper is vague about how to perform the translation from λ-style function definitions to these Prolog clauses.

Warren argues against having Prolog support either predicate variables or the λ-notation as primitives. He admits that he would prefer to use predicate variables in certain situations, but is concerned that overuse of such variables leads to confusing programs. As for λ-notation, he dislikes the effect that nesting of expressions can have on readability. In addition, he argues that his translation produces code that, when executed by a structure-sharing Prolog implementation, should be nearly as efficient as the "standard" way of representing functional objects:

> So, for DEC-10 Prolog at least, providing a specific implementation of higher-order objects would not produce huge efficiency gains over the approach I am advocating, and for most programs the overall improvement would probably be negligible. [War82, page 448]

3.4 Logic programming with equations

Equations can be considered as clauses of first-order logic with '=' as predicate symbol. Thus sets of equations can be viewed as logic programs. That is, given a set of equations E, it is possible to ask whether or not an equation e follows as a logical consequence of E, provided that the logic programming framework contains some notion of equality. Since equations can be used to define functions, the value of a functional expression can therefore be seen as a logical consequence of a set of function definitions. For example, if addition over terms in successor notation is defined as

$$0 + X = X,$$
$$s(X) + Y = s(X + Y),$$

evaluating $s(0) + 0$ is equivalent to finding a substitution $\theta = \{x/t\}$ such that $(s(0) + 0 = x)\theta$ is true. Unfortunately, many substitutions exist that satisfy the query. If the canonical (or completely reduced) value of $s(0) + 0$ is to be found, the logic programming system must employ some sort of special strategy for dealing with equality to find it.

Two main approaches have been used to include equality in logic programming. One method is to extend the inference mechanism so that it includes an inference rule for the equality relation. In other words, equality is given a special status, and the Prolog derivation mechanism is expanded such that it incorporates a strategy for finding canonical answers to equational queries. This was the approach taken by (G.) Robinson and Wos [RW69], and by Goguen and Meseguer [GM84]. Another technique is to treat equality like any other relation, and add to the theory a set of axioms that define equality. This second approach has been taken by Kowalski [Kow70], and more recently by van Emden and Yukawa [vEY87].

In [vEY87], two approaches were proposed for performing logic programming with equations. The standard set of axioms defining reflexivity, transitivity, and substitutivity forms a logic program, but it is computationally useless as its SLD-resolution search space contains many derivations yielding noncanonical answers and many infinite branches. Thus their first approach was to replace the axioms with an equivalent set whose SLD-derivation mimics rewriting without search. In this approach, the equations themselves form part of the logic program. Their second method obtains an efficient Prolog program by translating the equations to a set of Horn clauses not involving equality. Therefore, the equality axioms play no role in this second method and can be discarded.

Chapter 4

Translating λ-expressions to equations

Both of the methods to be presented for performing functional programming in logic require that function definitions be written as sets of equations. However, the λ-calculus is often a more concise way to describe functional expressions as it does not require that functions be given names. If λ-notation is to be supported by our systems, it will be necessary to perform a conversion from λ-expressions to sets of recursion equations.

Given a λ-abstraction of the form $\lambda x.\, B$, an equation containing the function can be constructed by giving the abstraction a name:

$$\text{introduced\_name} = \lambda x.\, B.$$

However, this equation is actually a mixture of the two formalisms, as it contains a λ-expression as right-hand side. What is needed is a translation that produces sets of λ-*free* equations.

The existence of such a translation seems plausible, as both notations can be used to describe the same functions. In fact, it would appear that Lisp provides both alternatives. For example, the function that squares its argument can be written as either of the following:

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

The first alternative looks like a named λ-expression

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

and the second looks like an equation

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

Both describe the same function, but one contains a λ and the other does not. At first glance, it looks as if it is possible to convert a λ into an application by moving it to the other side of the equation, and vice versa. In fact, it is the case that abstraction and application are interchangeable. For example, consider the expression $f = \lambda x.\, B$. If both sides of the equation are applied to the variable x, the left-hand side becomes $f : x$, and the right-hand side reduces to B. Thus the equation becomes $f : x = B$. Conversely, the original equation can be obtained by abstracting x out of both sides of $f : x = B$. This produces the equation $\lambda x.\, (f : x) = \lambda x.\, B$. The η-reduction rule of λ-calculus (see Section 2.1.5) allows the left-hand side to be reduced simply to f, resulting in the equation $f = \lambda x.\, B$

This interchangeability serves as a translation for named λ-abstractions of the form $f = \lambda x.\, B$, but is sufficient only if there are no further abstractions within B. However, the process can be repeated as many times as is necessary to remove all occurrences of λ from the original expression.

4.1 Representing λ-expressions in logic

Before a translation from λ-expressions to equations can be described in more detail, a representation in logic for λ-abstractions is required. When the λ-expression $\lambda x.\, B$ contains no free variables, it denotes a fully specified function. In other words, the value of the function depends only upon the value of the arguments required by the function. From the logic programming point of view, such an object can be named by a constant. If, on the other hand, the expression *does* have free variables, the function denoted depends on the values of the free variables; i.e. the values of the arguments alone are not enough to evaluate the function. Such a partially specified object must be named in logic by a term with the free variables as arguments. Hence, in the context of logic, the free variables of a λ-term are logic variables. They have a status different from that of the bound variables, which are λ-variables.

It would be convenient to represent bound variables by logic variables as well, but this presents a problem. As was mentioned in Section 2.1, bound variables have a *scope*. Different variables are allowed to share the same name as long as they occur in separate scopes. For example, x is used as a formal parameter twice in the following expression, but no confusion arises because each occurrence has its own scope.

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

Logic variables, on the other hand, do not include the notion of scope. Thus it would not be possible to use logic variables to represent x in the expression above as the logic variable X would denote the same value in each of the scopes. However, the names chosen for λ-variables have no semantical importance, and the meaning of *fourtimes* would not change if y was used instead of x throughout one of the scopes. The λ-calculus allows one to perform such renamings as long as they are done consistently (each occurrence within a given scope is changed), and the new name is distinct from

all other names in the current scope. A λ-expression in which all different variables have different names is said to be in *standard form.* If it is assumed that all λ-expressions are in standard form, it becomes possible to represent bound variables by logic variables.

So, in logic, a λ-abstraction is 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, $\lambda x.\, t : x$ must be written as $\lambda(x, t : x)$ according to the rules of logic syntax, but the traditional λ-calculus notation will be used for the purposes of discussion.

4.2 An example

As an example, let us show how to eliminate the λ's from the definition of the "twice" function, $t = \lambda f.\, \lambda x.\, f : (f : x)$. There are no free variables in the λ-expression, so the logic constant on the left-hand side of the equation is appropriate. The transformation described earlier in this chapter can now be used to remove the first occurrence of λ, giving $t : F = \lambda x.\, F : (F : x)$ where F is a logic variable. If the transformation is applied again, the equation becomes $t : F : X = F : (F : X)$. However, for reasons that will become clear in later chapters, it is preferred that there be no more than one application on the left-hand side of an equation. This can be achieved here by replacing the right-hand side obtained by a single λ-removal, $\lambda x.\, F : (F : x)$, by an appropriate logic term, such as $g(F)$. (Terms generated during the translation process, such as $g(F)$, are referred to as *introduced* function symbols.) A second equation is then created to define the term $g(F)$, resulting in the following pair of equations:

$$t : F = g(F)$$

$$g(F) = \lambda X.F:(F:X).$$

One further transformation and β-reduction on the second equation gives

$$t:F = g(F)$$
$$g(F):X = F:(F:X),$$

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. The transformation from λ-expressions to equations is due to M. Cheng, and first appears in [Che87], where it is formally defined as follows. In this definition, t is the λ-expression to be translated, and t' is the equivalent logic representation:

1. t' is t if t is a variable.
2. t' is $(t_1':t_2')$ if t is $(t_1:t_2)$.
3. t' is $f(y_1,\ldots,y_m)$ if t is $\lambda x.t_1$ and $FV(t) = \{y_1,\ldots,y_m\}$ where f is a new function symbol. The equation $f(y_1,\ldots,y_m):x = t_1'$ is then added.

4.3 Automating the translation

In general, automating the translation should be quite easy. It involves finding the free variables in a λ-expression, and including them as arguments to an introduced functor. Each time a new equation is created, it is regarded as a logic clause. Unfortunately, it is difficult to discuss programs for finding free variables and introducing new clauses without knowing something about the system in which one intends to implement the translation. For example, finding free variables requires searching

through an expression to find variables that are not bound by some outer abstraction. Since methods for disassembling terms are usually system-dependent, a general logic program to find free variables cannot be given. However, a program is shown in Appendix B that can be executed by ALS-Prolog to perform the desired translation.

4.4 Automating the reverse translation

It is not enough to be able to translate λ-expressions to equations. When function-valued expressions are evaluated, the result must be translated back into a form that is meaningful to the user. If, for example, the user submitted an expression that reduced to the "twice" function, it would be better to present the user with $\lambda a.\, \lambda b.\, a : (a : b)$ than t as an answer. Specifically, a program is desired that takes a function symbol occurring on the left-hand side of an equation and returns the λ-expression corresponding to the function denoted by the symbol. Formally, this operation can be defined by the following rules. (Here t' is the λ-expression reconstructed from the equational expression t.)

1. t' is t if t is a variable.

2. t' is $(t_1' : t_2')$ if t is $(t_1 : t_2)$.

3. t' is $\lambda z.\, t_1'$ if t is $f(s_1, \ldots, s_m)$, f is an introduced function symbol, and an equation $f(x_1, \ldots, x_m) : y = t_2$ exists. t_1 is obtained from $t : z$ by performing a single rewriting step.

Extensionality says that t is equivalent to $\lambda X.\, (t : X)$, provided X does not occur free in t. This is a useful equivalence because it allows the introduction of the term $t : X$, which appears as the left-hand side of one of the equations. Since $t : F = g(F)$,

the expression $\lambda X.\,(t:X)$ can be rewritten as $\lambda X.\,g(X)$. To summarize, extensionality was used to write t as an abstraction with $t:X$ as body. An equation was then used to replace $t:X$ by $g(X)$, so that it can be said that $t = \lambda X.\,g(X)$. But $g(X)$ is also a function symbol occurring on the left-hand side of an equation. Thus the process can be repeated on the term $g(X)$.

The following chain traces the construction of the λ-expression represented by the term t:

$$t = \lambda X.\,\underline{t:X} = \lambda X.\,\underline{g(X)} = \lambda X.\,\lambda Y.\,\underline{g(X):Y} = \lambda X.\,\lambda Y.\,X:(X:Y).$$

The first step in writing a logic program to perform the reverse translation is to axiomatize the knowledge that was used in moving from one step in the chain to the next. In the chain above, extensionality allowed t to be written instead as $\lambda X.\,t:X$. Thus our program must contain a clause describing this relationship:

$$X = \lambda Y.\,(X:Y) \leftarrow . \tag{1}$$

The second step in the chain, from $\lambda X.\,t:X$ to $\lambda X.\,g(X)$, was made possible by the use of an equation, which can be described by the following axiom:

$$\lambda Y.\,(X:Y) = \lambda Y.\,Z \leftarrow X:Y = Z. \tag{2}$$

The steps in the chain alternate between application of extensionality and the use of an equation, and the transformation is complete when it is no longer possible to find the current expression as the left-hand side of an equation. Thus the axiom

$$X = X \leftarrow \text{an equation does not apply to reduce } X. \tag{3}$$

must be added to describe the termination of the transformation. The condition on the axiom can be removed if it is guaranteed that the axiom will be used only if the others fail.

It is clear that axioms (1) and (2) can be combined to form

$$X = \lambda Y.Z \leftarrow X:Y = Z, \tag{4}$$

which is a consequence of extensionality and the use of an equation. This axiom describes one extensionality/equation cycle in the chain of equalities, but is not sufficient to describe the entire chain. This deficiency can be corrected by modifying the condition on axiom (4). When $X:Y = Z$ is combined with an axiom describing transitivity, $X = Y \leftarrow X = Z,\ Z = Y$, the condition can be rewritten as $X : Y = Z_1,\ Z_1 = Z$, resulting in a new axiom that describes a series of extensionality/equation steps:

$$X = \lambda Y.Z \leftarrow X:Y = Z_1,\ Z_1 = Z. \tag{5}$$

Thus a logic program for performing the reverse translation can be formed by combining this axiom (5) with the one describing termination (3):

$$X = \lambda Y.Z \leftarrow X:Y = Z_1,\ Z_1 = Z. \tag{6}$$

$$X = X \leftarrow . \tag{7}$$

When the equations resulting from the translation of a given λ-expression are added to these clauses, an SLD-derivation effecting the reverse translation can be obtained. In Figure 4.1, the term t is transformed back into its λ-calculus equivalent. It is assumed that the equations created in Section 4.2 have been added to axioms (6) and (7) to form a logic program. The initial query, $t = Z$, is resolved with axiom (6) to get the second list of goals. $t : Y = Z_1$ is then matched with the equation $t : Y = g(Y)$, yielding the third query. Next, axiom (6) is used again to get from the third goal statement, $g(Y) = Z_2$, to the fourth. The goal $g(Y) : Y_1 = Z_3$ is then resolved with an equation. And, finally, axiom (7) is used to complete the derivation.

This derivation could be performed by Prolog if it were possible to detect when the leftmost goal should be resolved with an equation, and when with one of the

$\leftarrow\ t = Z$

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

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

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

$\leftarrow\ g(Y) = Z_2$

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

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

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

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

reflexivity used

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

Figure 4.1: SLD-derivation that effects reverse translation

clauses from the logic program above. This problem can be solved by using a different predicate for equality in each case. Thus '=' will be used only in equations, and *eq* elsewhere:

$$eq(X, \lambda Y.\, Z) \leftarrow X : Y = Z_1,\ eq(Z_1, Z).$$
$$eq(X, X).$$

Attempts to translate variables are prevented by inserting `nonvar(X)`, so that the translation from equations to λ-expressions is effected by the following 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 (i.e. a defined function). For example, when the equations derived in Section 4.2 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. λ-terms of the form `lambda(U,lambda(U,...))` can never be generated because every instance of `lambda(Y,Z)` introduces new logic variables.

4.5 Results

In [Che87], a number of useful results are proven. It is shown that the equations generated by the translation in Section 4.2 are "well-behaved" in the sense of [HO82] (see Section 2.4), and that both the translation and its inverse as shown here terminate for a given expression. More importantly, it is proven that for a λ-term t, translating to equational form and back again results in a term that is equivalent to t (p. 55),

and that β-reduction can be simulated by translating an expression to equations, performing rewriting, and returning the result to λ-notation (p. 56). This last result is proven by induction on the number of λx abstractions in an expression M. It is shown that the reductions are the same for terms with zero and one abstraction, and that an expression with $k+1$ abstractions can be properly reduced to a term with only k abstractions.

4.6 Summary

This chapter has presented a translation from λ-expressions in standard form to sets of λ-free equations. In the process, a correspondence between logic variables and λ-variables was established, and a method for representing λ-expressions in logic was described. Prolog programs were also shown for performing both the translation and its inverse.

Chapter 5

Rewriting by resolution

Functional expressions can be evaluated by a term rewriting system if functions are defined by equations. (See Section 2.4.) The translation described in the last chapter makes it possible to automatically convert function definitions in λ-notation into sets of equations. Thus, the only ingredient missing from a scheme for functional programming in logic is a logic-based rewriting system.

Such systems have already been described in the literature [vEY87, CY86]. The approach taken in these papers was to create a logic program whose SLD-derivation followed the steps required to implement an algorithm for rewriting expressions. When rewrite rules were added as clauses to the logic program, it became possible to perform "Rewriting by Resolution". This chapter will analyze the program presented in [CY86] and show how it can be improved. The resulting program will then be modified to implement various reduction orders.

5.1 Introduction

Before describing "Rewriting by Resolution" in more detail, an illustration of the general rewriting process is needed. For the purposes of the example, rewriting will be performed over expressions built from the addition operator '+', and positive integers in successor notation, where addition is treated as a curried function of two arguments, i.e. $+ : s(0) : s(s(0))$. This family of expressions can be reduced by a rewriting system using the following rewrite rules:

$$+ : 0 : X = X$$
$$+ : s(X) : Y = s(+ : X : Y)$$

So, for example, they can be used to reduce $+ : s(0) : 0$ in these stages:

$$\underline{+ : s(0) : 0} \;=\; s(\underline{+ : 0 : 0}) \;=\; s(0).$$

Each reduction in this sequence is the result of applying a rewrite rule to some part of the current expression. The first step was to apply the rule $+ : s(X) : Y = s(+ : X : Y)$ to the expression $+ : s(0) : 0$, transforming it into $s(+ : 0 : 0)$. Here the rewrite rule was applied to the entire expression, but this will not always be the case. Often it is necessary to search for subexpressions that match the left-hand side of a rule, as was done in the second and final rewrite above. The reduction is finished when it is no longer possible to apply any of the rewrite rules to any part of the current expression.

5.2 Rewriting by resolution

The rewrite rules can be considered to be clauses in logic with '=' as a predicate symbol. If axioms describing transitivity, substitutivity, and reflexivity are now added, a system is created that is capable of rewriting by resolution. That is, a logic program is

transitivity	$X = Z$	$\leftarrow$	$X = Y,\ Y = Z$
substitutivity	$X : Y = X_1 : Y_1$	$\leftarrow$	$X = X_1,\ Y = Y_1$
	$s(X) = s(Y)$	$\leftarrow$	$X = Y$
reflexivity	$X = X$	$\leftarrow$	

Figure 5.1: Equality axioms

formed whose SLD-derivation effects the rewriting of expressions. Transitivity is necessary to continue the rewriting of an expression; substitutivity allows the rewriting of subexpressions, and reflexivity permits an expression to be rewritten to itself when no other axioms apply. One suitable set of equality axioms is shown in Figure 5.1.

Figure 5.2 shows how an SLD-derivation can be used to rewrite $+ : s(0) : 0$ to $s(0)$. The initial query $+ : s(0) : 0$=Z is resolved with the transitivity axiom from Figure 5.1, yielding the next list of goals. Then the goal $+ : s(0) : 0 = U$ is matched with the second equation in Section 5.1, resulting in the goal $s(+ : 0 : 0) = Z$. This goal is resolved with the transitivity axiom again, and the substitutivity axiom from Figure 5.1 is then used on the first goal in the new list, $s(+ : 0 : 0) = U$. Next, transitivity is applied to the goal $+ : 0 : 0 = V$, resulting in the list of goals shown in line six of the derivation. The first of these goals is resolved with the equation $+ : 0 : X = X$, giving the goals $0 = V$, and $s(V) = Z$. Since 0 cannot be further reduced, reflexivity is applied, leaving the goal $s(0) = Z$. As $s(0)$ is canonical, reflexivity is appropriate again, and the derivation is finished with $Z = s(0)$.

But the desired SLD-derivation is one of many possible derivations given the logic program described. For Prolog to perform the appropriate derivation, some sort of control information must be supplied regarding when to apply a given axiom. It was

$\leftarrow +:s(0):0=Z$

transitivity

$\leftarrow +:s(0):0=U,\ U=Z$

equation: $+:s(X):Y=s(+:X:Y)$

$\leftarrow s(+:0:0)=Z$

transitivity

$\leftarrow s(+:0:0)=U,\ U=Z$

substitutivity

$\leftarrow +:0:0=V,\ s(V)=Z$

transitivity

$\leftarrow +:0:0=U,\ U=V,\ s(V)=Z$

equation: $+:0:X=X$

$\leftarrow 0=V,\ s(V)=Z$

reflexivity

$\leftarrow s(0)=Z$

reflexivity

$\square$ with $Z=s(0)$

Figure 5.2: SLD-derivation that mimics rewriting

noted in [vEY87] that there is an algorithm that can be used to guide the reduction of an expression:

1. Check if the left-hand side of the leftmost goal is canonical. If it is, resolve the goal with the reflexivity axiom, and go to 1; otherwise resolve the goal with the transitivity axiom.

2. If possible, resolve the leftmost goal with one of the equations. If not, resolve it with a substitutivity axiom.

This algorithm requires that it be possible to determine if an expression is canonical (completely reduced) or not. Unfortunately, such a test is usually computationally expensive. The test was avoided in [CY86] by reversing the order of the two steps in the algorithm. Thus they assume that the expression is *not* canonical, and needs to be rewritten. A term is canonical only if it cannot be reduced. Thus, if it is impossible to use an equation or a substitutivity axiom, the current goal is resolved with the reflexivity axiom. This new algorithm can be described as follows:

1. Apply transitivity, followed by a successful rewrite to the leftmost goal and go to 1; or

2. Apply reflexivity to the leftmost goal if a rewrite is not possible.

Where by "successful rewrite" it is meant that the current expression is rewritten into something *different.*

The Prolog program in Figure 5.3 is given in [CY86] for performing rewriting by resolution. The clauses were tailored for the rewriting of functional expressions, and do not describe successor arithmetic. A second substitutivity axiom would have to be

```
eq1(X,Z,F) <--                  % transitivity
        eq2(X,Y,yes),
        eq1(Y,Z,F).
eq1(X,X,no).                    % reflexivity
eq2(X:Y,Z,F) <--                % substitutivity
        eq1(X,X1,F1),
        eq1(Y,Y1,F2),
        eq3(X1:Y1,Z,F3),        % function application
        or(F1,F2,F3,F).         % F is "no" if F1, F2, and F3 are "no"
eq2(X,Y,yes) <--
        X = Y.
eq3(X,Y,yes) <--
        X = Y.                  % using rewriting
eq3(X,X,no).                    % rewrite to itself

or(no,no,no).
or(no,yes,yes).
or(yes,no,yes).
or(yes,yes,yes).
```

Figure 5.3: Equality program from [CY86]

added to make this program capable of computing the example in the introduction:

```
eq2(s(X),s(Y),F) <-- eq1(X,Y,F).
```

A reduction is initiated by a call to `eq1(Expr,Reduced_Expr,_)`.

In this program, the test for canonicality has been avoided, but a new difficulty has arisen: it is now necessary to detect unsuccessful rewrites, as reflexivity must be used if an expression is irreducible. For an expression to have been rewritten, at least one of its subexpressions must have been reduced. (Or possibly the expression itself.) Thus, to tell if an expression has been rewritten, [CY86] uses a system of flags to record whether or not a rewrite has been performed at any level of the current expression. In effect, the third argument in the eq relations describes the composition of the first argument — the flag is yes if the first argument contains an expression found on the left-hand side of an equation.

5.3 A new approach

The flags in the previous program are an *ad hoc* measure, but are required because the eq2 relation does not properly describe successful rewrites without them. A successful rewrite is defined as one in which an expression is reduced. Hence the call `eq2(0,X)` should fail, as `0` is clearly canonical. But this call would succeed here with `X = 0` if it were not for the flags. In other words, eq2 defines canonical objects as expressions that rewrite to themselves, instead of as terms that *do not* rewrite. Thus, the flags are necessary to prevent infinite derivations in which a canonical expression is repeatedly rewritten to itself.

A better approach would be to use a relation that properly defines a successful rewrite in place of eq2. If a single application of a rewrite rule to an expression is

referred to as a *rewrite*, then the new relation should state that `eqB(X,Y)` is true if `X` reduces to `Y` in one or more rewrites. (In the development of the new program, `eqA` and `eqB` will be used to distinguish the new relations from those in the previous program.) Thus `eqB(X,Y)` would fail if `X` were canonical, as it would not be possible to find at least one rewrite rule that would reduce `X`.

Defining the new relation

In defining the new relation, there are three different cases that must be dealt with: rewriting the successor of an expression, rewriting a function application, and using a rewrite rule to reduce an expression. The first of these is straightforward. A term of the form $s(X)$ can be successfully rewritten to $s(Y)$ if X reduces to Y in one or more steps. This can be expressed by the following axiom:

$$eqB(s(X),\ s(Y)) \leftarrow eqB(X,\ Y). \tag{1}$$

Handling function application is more complicated, as rewriting an expression of the form $X:Y$ can be accomplished in one of two ways. Either X or Y (or both) must be reduced if the term $X:Y$ is to be reduced. Thus, either X is reduced to X' or Y is reduced to Y' in one or *more* steps. Once a successful rewrite has been performed on one of the subexpressions, there is no need to force a rewrite on the other. But since `eqB` is to perform one or more rewrites, an attempt will be made to rewrite the remaining subexpression. That is, if X has been reduced, then Y will be rewritten to Y' in zero or more steps, and vice versa. This can be expressed by the pair of clauses

$$eqB(X:Y,\ X':Y') \leftarrow eqB(X,\ X'),\ eqA(Y,\ Y') \tag{2}$$

$$eqB(X:Y,\ X':Y') \leftarrow eqA(X,\ X'),\ eqB(Y,\ Y'), \tag{3}$$

where $eqA(A, A')$ states that A rewrites to A' in zero or more steps. The only case remaining is the application of a rewrite rule. An expression X reduces to Y if there

is a rewrite rule of the form $X = Y$, hence the clause:

$$eqB(X,\ Y)\ \leftarrow\ X = Y. \tag{4}$$

The clauses defining eqA are similar to the `eq1` clauses, and incorporate the control specified by the revised algorithm. That is, either successfully rewrite X to get Y, and then rewrite Y to Z in zero or more steps, or use reflexivity. This information is described by the following pair of axioms:

$$eqA(X,\ Z)\ \leftarrow\ eqB(X,Y),\ eqA(Y,\ Z) \tag{5}$$

$$eqA(X,\ X)\ \leftarrow\ not(eqB(X,\_)). \tag{6}$$

When the eqB and eqA clauses are combined, a logic program for performing rewriting is obtained. Several simplifications can be made when this program is translated into Prolog: first, the condition on axiom (6) can be removed as long as (6) occurs after (5). Second, the first goal in the body of axiom (3) can be omitted if (3) appears after axiom (2). Clause (2) will only fail if it is not possible to successfully rewrite the first argument of the application, so it is not necessary to attempt rewriting it again in (3). Thus, axiom (3) can be replaced by the following clause:

$$eqB(X:Y,\ X:Y')\ \leftarrow\ eqB(Y,Y'). \tag{7}$$

Combining the new eqA and eqB clauses gives the Prolog program in Figure 5.4.

5.4 Reduction orders

The control specified in the new rewriting program results in a system that implements applicative order reduction. This is because an attempt is made to rewrite

```
eqA(X,Z) :-                    % X can be rewritten to Z if X
        eqB(X,Y),              % reduces to Y in 1 or more
        eqA(Y,Z).              % steps, and Y to Z in 0 or more.
eqA(X,X).                      % Otherwise X is canonical.

eqB(s(X),s(Y)) :-
        eqB(X,Y).
eqB(X:Y,X1:Y1) :-              % Rewrite X to X1 in 1 or more
        eqB(X,X1),             % steps, and Y to Y1 in 0 or more.
        eqA(Y,Y1).
eqB(X:Y,X:Y1) :-               % Or reduce Y to Y1 in 1 or more
        eqB(Y,Y1).             % steps.
eqB(X,Y) :- X=Y.               % Apply a rewrite rule.
```

Figure 5.4: New rewriting program

both the operand and the operator before trying to use a rewrite rule. It is interesting to observe the various reduction orders that can be achieved through minor modifications of this program. For example, if the eqB goals are reordered such that an equation is tried before any arguments are rewritten, a system is created that performs reductions in an order somewhere between normal and applicative. (See Figure 5.5.) The only thing that keeps this version from implementing completely normal order reduction is the fact that *all* arguments are reduced if an outermost reduction cannot be performed. Normal order can be achieved by removing the eqA goal in the third eqB clause, yielding the program in Figure 5.6. Now, if an equation does not apply to a function application, the subexpressions on either side of the : are rewritten individually, from left to right, as required. Notice that the meaning of eqB has changed. Instead of specifying one or more rewrites as it did in the previous programs, it now requires exactly one rewrite.

```
eqA(X,Z) :-
        eqB(X,Y),
        eqA(Y,Z).
eqA(X,X).

eqB(X,Y) :-                    % Try to reduce the outermost
        X=Y.                   % redex first.
eqB(s(X),s(Y)) :-
        eqB(X,Y).
eqB(X:Y,X1:Y1) :-              % If it doesn't work, then try
        eqB(X,X1),             % rewriting both X and Y.
        eqA(Y,Y1).
eqB(X:Y,X:Y1) :-
        eqB(Y,Y1).
```

Figure 5.5: Rewriting system implementing mixed-order evaluation

```
eqA(X,Z) :-
        eqB(X,Y),
        eqA(Y,Z).
eqA(X,X).

eqB(X,Y) :-                    % Try to reduce the outermost
        X=Y.                   % redex first.
eqB(s(X),s(Y)) :-
        eqB(X,Y).
eqB(X:Y,X1:Y) :-               % If it doesn't work, then
        eqB(X,X1).             % try reducing X.
eqB(X:Y,X:Y1) :-
        eqB(Y,Y1).             % Or reduce Y.
```

Figure 5.6: Rewriting system implementing normal-order evaluation

5.5 Summary

This chapter has focused upon the development of a Prolog program whose SLD-derivation effects term rewriting. Previous rewriting approaches were analyzed, and an improved version of the program presented in [CY86] was formulated. Several variations were shown which perform reduction in orders varying from applicative to normal-order. Since functional programming can be implemented by term rewriting, these logic-based systems allow functional expressions to be evaluated in Prolog.

Chapter 6

Compilational approach

In the last chapter, a rewrite-based functional programming system was proposed. The efficiency of the system, while surprisingly good for a system implemented in Prolog, leaves room for improvement. (See benchmarks in Appendix A.) Like all rewriting systems, it spends the majority of its time searching for subexpressions to rewrite.

Expressions contain information, when precedence rules are taken into account, about the order in which their subexpressions must be evaluated. However, this information is not discovered by a rewriting system until a computation is underway. Thus an optimization would be to find the subexpressions requiring evaluation ahead of time, and determine the order in which they must be computed. Once the required computations have been identified, they can be translated into Prolog, allowing expressions to be evaluated in logic. This analysis and translation can also be used to help compile the equations themselves into equivalent logic clauses, so that it becomes possible to perform efficient functional programming in logic without the use of the term rewriting system.

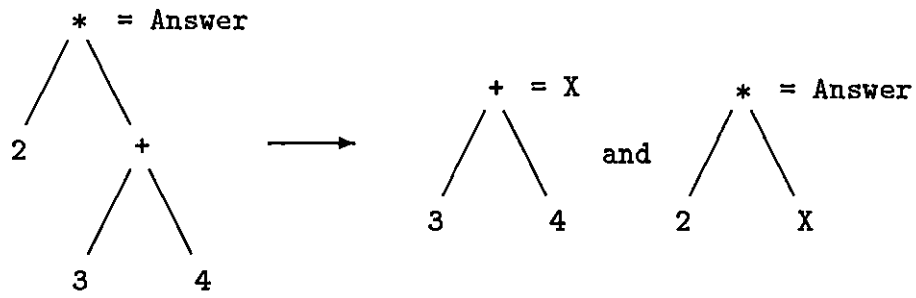

Figure 6.1: Disassembling an expression

6.1 Introduction

As an illustration, imagine trying to evaluate the expression `2*(3+4)`. It is clear that the term `3+4` must be evaluated before the multiplication can be performed. Thus one could first have the computer solve `3+4 = X`, and then use the result in calculating `2*X = Answer`. So it is possible to disassemble expressions and arrange the subexpressions so that performing a series of primitive computations results in the evaluation of the original term[1]. (See Figure 6.1.) If all expressions were "flattened" in this manner before evaluation, a simpler rewrite system could be used to compute their values, as it would not be necessary to search for subexpressions to be rewritten.

But once expressions have been divided into their basic subcomputations, there is little reason to use a rewriting system of any kind — the individual computations can be translated directly to Prolog instead. For example, `3+4 = X` can be expressed as the relation `sum(3,4,X)`, and `2*X = Answer` as `times(2,X,Answer)`. Thus the

[1]Note that the order in which the calculations are performed is important. The discussion in this chapter will assume that applicative order evaluation is to be performed, and will order subexpressions accordingly.

original expression could be computed by following query[2].

```
?- sum(3,4,X), times(2,X,Answer).
```

6.2 Translating equations to relational form

The technique described above was used to break an expression into individual relations so that its value could be computed by Prolog. This procedure is utilized by a process called *relationalization* [vEM81] to translate the equations themselves into logic clauses.

The relationalization axiom

$$\forall x_1, \ldots, x_n, v \quad f(x_1, \ldots, x_n) = v \leftrightarrow ff(x_1, \ldots, x_n, v) \tag{1}$$

states that ff relates the value of the function f to its arguments. This axiom represents a family of such axioms, one for each function f to be relationalized. For example, the first transformation in the previous section can be justified by the following axiom:

$$\forall x, y, z \quad x + y = z \leftrightarrow sum(x, y, z). \tag{2}$$

And the relationalization of function application is described by this axiom:

$$\forall x, y, z \quad x : y = z \leftrightarrow apply(x, y, z). \tag{3}$$

Since all of the equations generated by the translation described in Chapter 4 have left-hand sides of the form $x : y$, axiom (3) can be used to convert the equations

[2]The reader may wonder why the relational form is used here to describe arithmetic when the introduction to this thesis favored the `is` built-in predicate. It is useful to think of the individual subcomputations as representable by relations, as the extensions necessary for function application require the relational notation.

to clauses. But a problem is encountered when trying to relationalize an equation whose right-hand side contains an evaluable expression. The z in axiom (3) refers to the value of the right-hand side. Thus, in cases where the right-hand side requires evaluation, the expression must be broken into subexpressions and arranged such that the value of the right-hand side is computed and used as z.

As an example of the relationalization process, consider transforming the equation $g(F):X = F:(F:X)$ to relational form. The equation states that the value of $g(F):X$ is the same as the value of $F:(F:X)$. In other words, $g(F):X = V$ if and only if $F:(F:X) = V$. The relationalization axiom for application allows the left-hand side of this equation to be written as $apply(g(F), X, V)$, but this relation is only true if $F:(F:X) = V$. Thus the complete relation is described by the following clause:

$$apply(g(F), X, V) \leftarrow F:(F:X) = V.$$

The condition here can be flattened, as was done in the introduction to this chapter, yielding a pair of equations: $F:X = U$, and $F:U = V$. When these equations are relationalized, the clause above can be written as

$$apply(g(F), X, V) \leftarrow apply(F, X, U),\ apply(F, U, V),$$

which is the relationalized version of $g(F):X = F:(F:X)$. (Note that the clause produced by this process is identical to the one used by Warren in Section 3.3.)

6.3 Our approach to relationalization

A Prolog program is given in [vEM81] for relationalizing equations, but it is unjustified, and a poor example of logic programming. [vEY87] proves the correctness of the translation and describes an improved method for performing the conversion, but

does not attempt automation. Our approach is based on the observation that the relationalization axioms, in conjunction with axioms of equality, can be used to derive from an equation its relationalized form. The remainder of this chapter will formalize such proofs and automate them, so that relationalized clauses can be obtained directly from their proofs.

An informal proof can be constructed by translating the left-hand side of an equation to its relational version, and forming a chain of implications showing that the relation holds only for the relationalized form of the right-hand side. As an example, the equation used above can be converted to relational form as follows: First, translate the left-hand side, $g(F) : X$, to $apply(g(F), X, Z)$. Now form the following chain of implications:

$$apply(g(F), X, Z) \leftarrow g(F) : X = Z \leftarrow g(F) : X = U,\ U = Z \leftarrow F : (F : X) = Z$$
$$\leftarrow F : X = U,\ F : U = Z \leftarrow apply(F, X, U),\ apply(F, U, Z).$$

The implications are justified, respectively, by relationalization axiom (3), transitivity, the equation, substitutivity, and relationalization again. The chain proves

$$apply(g(F), X, V) \leftarrow apply(F, X, U), apply(F, U, V),$$

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

Deriving the relationalized clause in this manner assures us that the result is correct, but it is also important for our purposes that the transformation be automatable. The chain of implications suggests that it is possible to have Prolog create such a proof, but the chain must first be formalized into an SLD-derivation. This requires precise definitions of the axioms used as they are the input clauses of the SLD-derivation.

The derivation in Figure 6.2 could be performed by Prolog if two obstacles were

$\leftarrow \ apply(g(F), X, Z)$

$apply(X, Y, Z) \ \leftarrow \ X : Y = Z$

$\leftarrow \ g(F) : X = Z$

$X = Z \ \leftarrow \ X = Y, \ Y = Z$

$\leftarrow \ g(F) : X = Z_1, \ Z_1 = Z$

$g(F) : X = F : (F : X)$

$\leftarrow \ F : (F : X) = Z$

$X : Y_1 = Z \ \leftarrow \ Y_1 = Y_2, \ Y_2 = Z$

$\leftarrow \ F : X = Z_2, \ F : Z_2 = Z$

$X : Y = Z \ \leftarrow \ apply(X, Y, Z)$

$\leftarrow \ apply(F, X, Z_2), \ F : Z_2 = Z$

$X : Y = Z \ \leftarrow \ apply(X, Y, Z)$

$\leftarrow \ apply(F, X, Z_2), \ apply(F, Z_2, Z).$

Figure 6.2: SLD-derivation of "twice" clause

overcome: First, the derivation is but one of many possible derivations given the input clauses shown. Prolog would require some sort of control information to ensure that the appropriate derivation were performed. Second, and more importantly, the SLD-derivation is incomplete. Prolog derivations, when they terminate, are expected to succeed or fail. The derivation in Figure 6.2 does neither. It is as if a normal derivation were frozen before completion, yielding a "conditional answer" [Vas86, vE88]. That is, a clause whose head is the initial goal with the cumulative substitution applied to it, and whose body is the final goal statement in the derivation. The derivation proves that the conditional answer is logically implied by the input clauses of the SLD-derivation. Before the translation can be automated, both the conditional answer and control problems must be addressed.

6.3.1 Computing conditional answers

Derivations yielding conditional answers are identical to normal SLD-derivations, except that they are not allowed to terminate. Instead, the derivation must be stopped when the desired conditional answer has been obtained. Thus it makes sense to use a meta-interpreter to produce conditional answers, as this allows the basic derivation mechanism of Prolog to be used. A general meta-interpreter yielding conditional answers is shown in Figure 6.3. The first clause describes the trivial conditional answer: a goal is true if it is true. The next clause states that the conditional answer `Concl` ← `Cond` holds if a goal is selected from the list of goals `Concl` and resolved with a clause from the database, yielding a new list of goals `Cond`. 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 allows a more general goal-selection scheme than Prolog in that it is possible to choose a goal other than the leftmost.

```
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.
```

Figure 6.3: Conditional answer meta-interpreter

6.3.2 Adding control information

The meta-interpreter in Figure 6.3 gives many different conditional answers. To compute exactly the one desired requires appropriate definitions for `decompose` and `clause`. Notice that the computation strategy used in the derivation in Figure 6.2 is quite straightforward. There are several steps that will always be performed at the beginning of any derivation that proves a relationalized equation:

1. Apply an axiom to unrelationalize the initial goal,
2. Resolve the resulting equation with the transitivity axiom,
3. Make use of the equation to be relationalized.

Thus these initial steps can be programmed into the meta-interpreter. Then, as long as there are still goals with = as predicate symbol, they are selected and relationalized if possible. It may be that the expression on the left-hand side of the = requires flattening before relationalization can be performed. If this is the case, the goal should be resolved with an appropriate substitutivity axiom. When no more such

goals are found, the derivation stops and returns the computed conditional answer. A more sophisticated meta-interpreter incorporating this control information can be found in Appendix C.

6.4 Computing with relationalized equations

Once a set of equations has been translated to relational form, they can be used to efficiently compute higher-order functional expressions in Prolog. Any expression to be evaluated can be relationalized into a list of goals that, when posed to Prolog as a query, will compute the value of the expression. For example, *twice*:*succ*:0 would be translated into a pair of goals, forming the query:

```
?- apply(twice,succ,X), apply(X,0,Answer).
```

The result of computing a function-valued expression will be an introduced functor, and it can be transformed back into the corresponding function in λ-notation by the program in Section 4.4. For example, evaluating *twice*:*succ* would result in the term $g(succ)$, which is equivalent to the function `lambda(X,succ:(succ:X))`.

6.5 Summary

Expressions can be broken up into easily-computable subexpressions and ordered such that their evaluation effects the computation of the original expression. The subcomputations resulting from disassembling an expression in this manner can be translated to Prolog relations, allowing arithmetic expressions to be computed by Prolog. This procedure is employed by a process called *relationalization* to translate

equations to logic clauses, allowing efficient evaluation of functional expressions in logic.

This chapter presented a method for automating the translation from equation to relationalized clause such that the resulting clause is obtained directly from its *proof*. The proof is in the form of an incomplete SLD-derivation, or *conditional answer*. Thus, the program for deriving the relationalized version of an equation is a specialized meta-interpreter that collects the appropriate conditional answers.

Chapter 7

Conclusions

This thesis has presented a pair of methods for evaluating higher-order functional expressions in logic. It was shown in Chapter 4 that any expression in λ-notation can be translated into a set of equations. The resulting equations can then either be used as rewrite rules by a logic-based term rewriting system (Chapter 5), or compiled to an equivalent set of directly-executable Prolog clauses (Chapter 6). The relationship between λ-expressions, equations, and these two functional programming schemes is summarized in Figure 7.1.

It should be noted that the results of relationalizing the "twice" example are identical to the clauses used by Warren in [War82] to compute the same expression in Prolog. However, it is impossible to tell if our translation is the same as his, as [War82] does not fully describe his technique.

One of the benefits of both of our approaches is that function-valued expressions can be returned to the user in λ-notation. This is in contrast to Lisps, which do not support λ-notation, and are therefore unable to report the result of a function-valued computation.

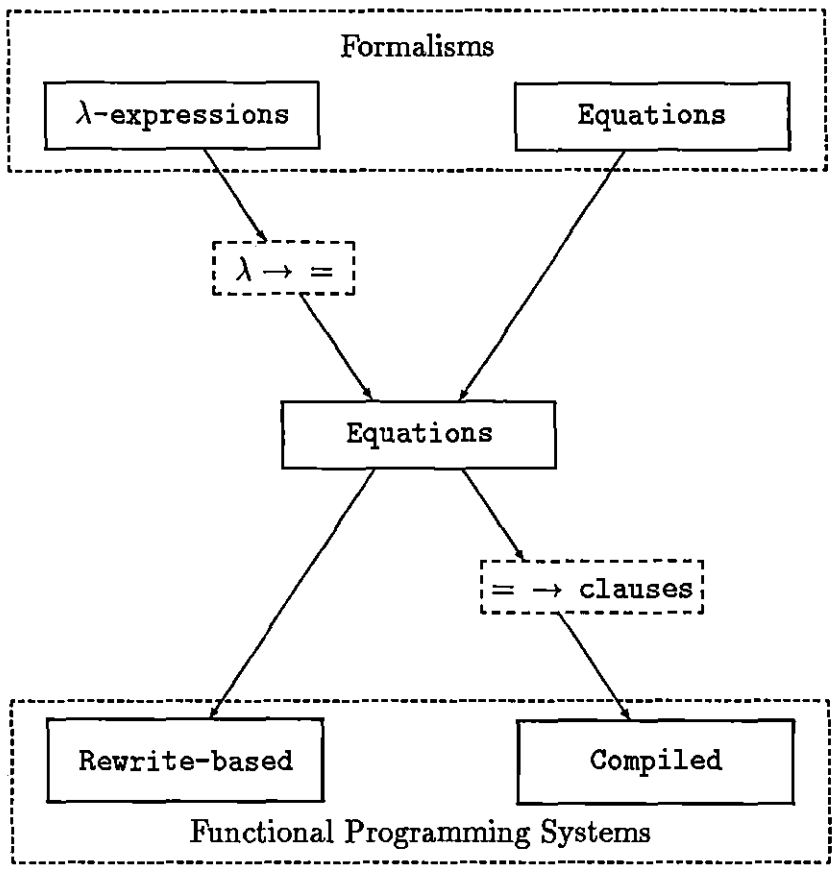

Figure 7.1: Relationship between λ's, equations, and our approaches

Benchmark tests were performed to assess the efficiency of the functional programming schemes presented in this thesis, and the results are promising. (Detailed information on the tests can be found in Appendix A). In fact, the "twice" example was computed several times faster by our compilational system than by the faster of the two Lisp implementations that participated in the tests. However, the Prolog in which our systems were written also proved to be several times faster than the Lisps on general list-processing tasks, complicating an interpretation of the results. Though it does not make sense to directly compare the times required to evaluate the functional benchmark in light of the list-processing differences, the performance of the Prolog-based systems is quite impressive.

The contributions in this thesis are:

- Prolog programs for automatically performing the translation from λ-expressions to equations, and from introduced function symbols back to λ-expressions were shown.
- An improved logic-based rewriting system was derived. Variations were also shown that perform reductions in orders ranging from applicative to normal-order.
- A method for automatically deriving relationalized clauses from their proofs was demonstrated.
- Benchmarks were performed that tentatively indicate that functional programming in logic is competitive with Lisp implementations for evaluating higher-order functional expressions.

7.1 Future work

Our experience indicates that there are several areas that need to be more fully investigated:

Mixing equations and λ-notation: Both of the methods proposed for performing functional programming in logic require that functions be defined by equations. The translation presented in Chapter 4 makes it possible to start with λ-notation, but there is no reason why the user could not define functions with equations directly, or mix equations and λ-expressions. It would seem that the two complement each other, and more work should be done to determine how best to use a combination of the formalisms.

Lazy evaluation: A rewriting system was shown that performs normal-order evaluation, but it is mainly of theoretical interest as plain normal-order evaluation is quite inefficient. A logic-based system that implements fully lazy evaluation would be much more powerful and useful.

Compiling different reduction orders: The rewriting approach permits different reduction orders, but the compilational system performs only applicative-order evaluation. Further work should be done in trying to similarly compile other reduction orders.

Virtual machines: Only a small subset of the Prolog language is used in computing with the clauses resulting from relationalization. If the required WAM instructions were isolated, it might be that an optimized virtual machine could be constructed for evaluating functional expressions.

Including functions in Prolog: This thesis has made no mention of how functional programming capabilities should be integrated into Prolog. One solution,

that would not require any significant changes to the Prolog framework, would be to allow functional expressions as the second argument to the `is` built-in predicate. However, more research should be done to determine how best to interface the two paradigms, and what sort of syntax should be used to denote functional expressions.

Bibliography

[AKN86] Hassan Aït-Kaci and Roger Nasr. Residuation: a paradigm for integrating logic and functional programming. Technical Report AI-359-86, MCC, 1986.

[Che87] M.H.M. Cheng. *Lambda-equational Logic Programming.* PhD thesis, University of Waterloo, 1987.

[CY86] Mantis H.M. Cheng and Keitaro Yukawa. Ap: an assertional programming system. Technical Report CS-86-11, University of Waterloo, 1986.

[GM84] J.A. Goguen and J. Meseguer. Equality, types, modules, and (why not?) generics for logic programming. *JLP*, 2:179–210, 1984.

[HO82] C.M. Hoffman and M.J. O'Donnell. Programming with equations. *ACM Transactions on Programming Languages and Systems*, 4(1):83–112, 1982.

[HS86] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

[Hug84] R.J.M. Hughes. *The Design and Implementation of Programming Languages.* PhD thesis, Programming Research Group, Oxford, 1984.

[Joh85] T. Johnsson. Lambda lifting: transforming programs to recursive equations. In J.-P. Jouannaud, editor, *Functional programming languages and com-*

puter architecture, pages 190–203. Springer-Verlag Lecture Notes in Computer Science 201, 1985.

[Jon87] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* Prentice-Hall, Englewood Cliffs, New Jersey, 1987.

[Ken82] J.R. Kennaway. The complexity of a translation of lambda calculus to combinators. Technical report, Department of Computer Science, University of East Anglia, 1982.

[Kow70] R.A. Kowalski. The case for using equality axioms in automatic demonstration. In M. Laudet et al., editor, *Symposium on Automatic Demonstration*, volume 125, pages 112–127. Springer-Verlag, New York, New York, 1970. Lecture Notes in Mathematics.

[McC60] J. McCarthy. Recursive functions of symbolic expressions and their computation by machine. *Comm. ACM*, 3:184–195, 1960.

[Nar88] Sanjai Narain. Log(f): an optimal combination of logic programming, rewriting, and lazy evaluation. Technical report, The RAND Corporation, 1988.

[O'D77] M.J. O'Donnell. *Computing in Systems Described by Equations.* Lecture Notes in Computer Science 58. Springer-Verlag, 1977.

[Rog67] Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[RW69] G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, pages 135–150. American Elsevier, New York, 1969.

[Tur79] D.A. Turner. A new implementation technique for applicative languages. *Software Practice and Experience*, 9:31–49, 1979.

[Tur82] D.A. Turner. Recursion equations as a programming language. In J. Darlington, P. Henderson, and D. Turner, editors, *Functional Programming and its Applications*, pages 1–22. Cambridge University Press, 1982.

[Vas86] P. Vasey. Qualified answers and their application to transformation. In *Proceedings of the Third International Logic Programming Conference*, pages 425–432, 1986.

[vE88] 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.

[vEM81] 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.

[vEY87] M.H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 4:265–288, 1987.

[War82] 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.

Appendix A

Benchmarks

Designing and implementing benchmark tests is often difficult work. One must try to create fair tests that, as much as possible, test only a single aspect of performance. Then, once the results of the tests are in, one must be careful not to draw unfounded conclusions from them. But properly designed benchmarks can serve as a useful comparison. In this chapter, an attempt will be made to compare both of the functional evaluation strategies described in this thesis to existing functional languages.

There are, of course, many different aspects of these systems that could be compared. One of the most important and easily measured of these is execution speed. Speed is of primary importance here because both Lisp and Prolog require garbage collection in order to fit into a reasonable amount of memory, and poor garbage collection schemes translate into slower execution speeds. In particular, the speed at which the systems can evaluate expressions involving higher-order functions is of interest, as Prolog is already able to evaluate simple arithmetic expressions.

The languages

Two different Lisp implementations were used as examples of functional programming systems, primarily because they were readily available to us. They were Franz Lisp (Opus 38.91) and IBUKI Common Lisp (Release 01/01). The logic-based systems were implemented in ALS-Prolog (Version 1.01). Both of the lisp implementations could be used as interpreters or compilers, and the benchmarks were evaluated in each mode.

The tests

As was mentioned in Section 2.2, computing multiple applications of the "twice" function makes a good test of higher-order function processing, and has been used as a benchmark since at least [Tur79]. Thus, the evaluation of the expression

$$((((twice : twice) : twice) : twice) : succ) : 0$$

was chosen as the first benchmark. (*twice* and *succ* are defined in Figure 2.1.) But even if a Prolog-based system were to perform this computation faster than the Lisps, the possibility remains that a fast Prolog might do *everything* better than a slow Lisp. Hence, two more tests were performed that were designed to compare the list processing abilities of the Lisps and ALS-Prolog: naive reversal of a nested list, and the creation of all permutations of a given list. It is hoped that the results of these last two tests will provide more of a baseline against which the results of the higher-order functional evaluation benchmark can be compared.

A.1 Technical foreword

All tests were executed on a Sun 3/280. The resolution of the timer in each language was $1/60^{th}$ of a second. Since some of the tests being measured took only a few

hundredths 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[1]. 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 the Lisps were of different dialects turned out to be only a minor factor. The largest difference between the two had to do with creating the closures necessary for implementing the "twice" benchmark. The list-processing tests are, in fact, identical except for the calls used to read the internal clock. Thus only one Lisp program will be presented for each of the last two tests.

A.2 The "twice" benchmark

A.2.1 Prolog notes

The functional expression above was evaluated by both the rewrite-based and compiled systems. The compiled program consisted of the following clauses:

```
twice:F = g(F).
g(F):X = F:(F:X).
succ:X = Y :- Y is X+1.
```

And the computation was initiated by the query

```
?- apply(twice,twice,U), apply(U,twice,V), apply(V,twice,W),
                                    apply(W,succ,X), apply(X,0,Result).
```

[1]Measuring the time to complete 50 iterations of a given test reduces the smallest measurable time interval to .0003 seconds, but the variability between test runs was closer to .001 seconds.

The rewrite-based approach was represented by both the applicative and mixed-order rewrite programs from Chapter 5 (Figures 5.4 and 5.5). Unfortunately, these systems were unable to compute the benchmark without a slight modification that changed the way in which they used memory: a cut was inserted after the first goal in the first `eqA` clause in each program so that the systems were able to eliminate unnecessary choice-points.

A.2.2 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)
) )
```

A.2.3 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)
) )
```

A.3 The "naive reverse" benchmark

This is a slight modification of the traditional "naive reverse" benchmark. Here indefinite nesting of lists is allowed 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)))))
```

A.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).
```

A.3.2 Lisp code

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 naive 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, but an equivalent optimization to the Prolog code, the use of difference-lists, was not implemented.

```
(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)))
)
```

A.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 a list of the permutations of the first n elements exists. To get the list for $n+1$ elements, the new element is inserted in each possible place in each list.

A.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).
```

A.4.2 Lisp code

The Lisp code mirrors the Prolog implementation.

Interpreted Lisps		*Compiled Lisps*		*Prolog-based systems*		
IBUKI	Franz	IBUKI	Franz	Rewrite(a)	Rewrite(m)	Compiled
67.0	25.6	7.5	19.7	68.8	21.2	1.5

Table A.1: Results of the "twice" benchmark

```
(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))
) ) )
```

A.5 Table of results

The results of the benchmarks are summarized in tables A.1 and A.2. All timings are in seconds. In Table A.1, Rewrite(a) stands for the applicative order program, and Rewrite(m) denotes the mixed order variant.

Test	*Interpreted Lisps*		*Compiled Lisps*		*Prolog*
	IBUKI	Franz	IBUKI	Franz	
reverse	1.800	0.656	0.078	0.157	0.023
permutation	1.973	0.857	0.383	0.382	0.065

Table A.2: Results of the list-processing benchmarks

Appendix B

Code for translating λs to equations

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

%       translate(Xs,Ys) holds if Ys is a list of lambda-free
%  expressions equivalent to the expressions in the list Xs.  As
%  lambda expressions in Xs are found and translated to equations,
%  new clauses are created and asserted.
%       As an example, translate([lam(X,X+Y)],[g(Y)]) is true, and
%  would result in the assertion of the clause "g(Y):X = X+Y".

translate([],[]).
translate([X|Xs],[X|Ys]) :-              % ignore variables
        var(X),
        translate(Xs,Ys).
translate([X|Xs],[X|Ys]) :-              % ignore atoms
        atomic(X),
        translate(Xs,Ys).
translate([lam(X,Y)|Xs],[N|Ys]) :-
        equivalent(lam(X,Y),N),          % process lambda expressions
        translate([Y],[FinalY]),         % try to process body
        assert(N:X = FinalY),
```

```
        translate(Xs,Ys).
translate([X|Xs],[NewX|Ys]) :-
        nonvar(X),
        X \== lam(_,_),
        X =.. [Functor|Args],          % decompose a term
        translate(Args,NewArgs),       % process the args
        NewX =.. [Functor|NewArgs],    % put the term back together
        translate(Xs,Ys).

%       equivalent(X,Y) is true if Y is an introduced functor that
%  is equivalent to the lambda expression X.  That is, Y will
%  contain the free variables in X.

equivalent(lam(X,Y),Z) :-
        varlist(lam(X,Y),Vars), % find the free vars
        unused_name(NewName), % get an unused name
        Z =.. [NewName|Vars]. % create the functor

%
% varlist( T, L ) holds if L is a list all occurences of distinct
%       variables in term T
%
%       written by: Mantis H.M. Cheng (Oct20/87)
%
varlist( X, L ) :-
        varlist( X, [], [], L ),
        ! .

%
% auxiliary predicates of varlist/2
%
varlist( X, Bound, FreeSoFar, NewFree ) :-
        var( X ),
        free_var( X, Bound, FreeSoFar, NewFree ).
varlist( X, _, L, L ) :-
        atomic( X ),
        ! .
varlist( [], _, L, L ) :-
        ! .
```

```
varlist( T, Bound, FreeSoFar, NewFree ) :-
        T =.. [lam,X,Y],
        var( X ),
        join_var( X, Bound, NewBound ),
        varlist( Y, NewBound, FreeSoFar, NewFree ),
        ! .
varlist( T, Bound, L0, L ) :-
        T =.. [F|A],
        !,
        varlist1( A, Bound, L0, L ) .
varlist( [A|T], Bound, L0, L ) :-
        varlist1( [A|T], Bound, L0, L ) .

%
% auxiliary predicates of varlist4/
%
varlist1( [], _, L, L ) .
varlist1( [T|A], Bound, L0, L ) :-
        varlist( T, Bound, L0, L1 ),
        varlist1( A, Bound, L1, L ) .


%       join_var is an auxiliary predicate of varlist4/ that adds a
%  variable X to a list of unique variables without instantiating it
%  or unifying it with anything.

join_var( X, [], [X] ) :-
        ! .
join_var( X, [Y|Z], [Y|Z] ) :-
        X == Y,
        ! .
join_var( X, [Y|Z], [Y|Z1] ) :-
        join_var( X, Z, Z1 ) .

%       free_var(V, Bs, Fs, NewFs) holds if NewFs is the list of
%  variables obtained by possibly adding V to the list Fs.  Bs and
%  Fs are expected to be lists of bound and free variables,
%  respectively.  If V occurs in the list of bound vars, then it is
%  not to occur in NewFs.  Otherwise, it is added to Fs, yielding NewFs.
```

```
free_var( Var, Bound, Free, Free ) :-
        varmember(Var, Bound ).
free_var( Var, Bound, Free, NewFree ) :-
        non_varmember( Var, Bound ),
        join_var( Var, Free, NewFree ).


%       varmember and non_varmember are variants of member and
%  non_member that work with lists of variables, as they are careful
%  not to instantiate the list elements.

non_varmember(X,[]).
non_varmember(X,[Y|Ys]) :-
        X \== Y,
        non_varmember(X,Ys).

varmember(X,[Y|Ys]) :-
        X == Y.
varmember(X,[Y|Ys]) :-
        X \== Y,
        varmember(X,Ys).

%       unused_name instantiates its argument to a unique atom
%  formed by appending a number to the string "introduced_".
%  When the translation routine is first run, the last used
%  number is set to zero.  Calls to unused_name increment the
%  last used number before using it, and store the result.

last_num(0).

unused_name(Name) :-
        last_num(X),            % find last used number
        Num is X + 1,           % increment it
        retract(last_num(_)),
        asserta(last_num(Num)), % store it
        name(Num,NumString),
        append("introduced_",NumString,String),
        name(Name,String).
```

Appendix C

Relationalization meta-interpreter

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

%     condAnswer(Conc,Cond) is true if the list of goals Cond can
%  be derived from the initial goal Conc.  The derivation uses the
%  goal selection strategy described by "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( g(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]).
```

Vita

Surname: Richards | Given Names: Bradley Eric

Place of Birth: Ames, Iowa | Date of Birth: August 4, 1966

Educational Institutions Attended:

Gustavus Adolphus College	1984 to 1988
University of Victoria	1988 to 1990

Degrees Awarded:

B.A.	1988	Gustavus Adolphus College, Minnesota

Partial Copyright License

Title of Thesis:

Contributions to Functional Programming in Logic

Author: [redacted]
Bradley E. Richards
April 19, 1990