COMPILING A FUNCTIONAL LANGUAGE TO PROLOG

by

Csaba Csáki
Master of Engineering
Technical University of Budapest, 1988

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

ACCEPTED
FACULTY OF GRADUATE STUDIES
[redacted]
DEAN
DATE 28 Sept 93

in the Department of Computer Science

[redacted] accept this thesis as conforming
to the required standard

[redacted]
Dr M H M Cheng, Supervisor (Dept of Computer Science)

[redacted]
Dr M H van Emden, Dept Member (Dept of Computer Science)

[redacted]
Dr M R Levy, Dept Member (Dept of Computer Science)

[redacted]
Dr D S Parker, External Examiner (Univ of California at Los Angeles)

Supervisor Dr M H M Cheng

ABSTRACT

The two main paradigms of declarative computation are logic programming and functional programming. There are problems which can be expressed more naturally in a logic language than in a functional language and vice versa. Therefore, the integration of the two paradigms is not only of theoretical but of practical interest as well.

In this thesis we discuss the introduction of functional expressions into logic programs. We present a method for translating the expressions into equations and for translating the equations into logic clauses. The resulting code is an ordinary Prolog program and can be efficiently executed using conventional implementation techniques.

Examiners

[redacted]

Dr M H M Cheng, Supervisor (Dept of Computer Science)

[redacted]

Dr M H van Emden, Dept Member (Dept of Computer Science)

[redacted]

Dr M R Levy, Dept Member (Dept of Computer Science)

[redacted]

Dr D S Parker, External Member (Univ of California at Los Angeles)

Table of Contents

List of Figures

Acknowledgment

I would like to thank my supervisor, Dr. M. H. M. Cheng for his continuous encouragement and support during my thesis research. Despite his busy time schedule he let me drop into his office with my problems anytime. He was always willing to share his knowledge and expertise.

I would also like to express my gratitude to Dr. M. H. van Emden, who lead my first steps in academic research. He was a source of great inspiration whenever I encountered any difficulty with expressing my ideas. Lots of credit goes to him for his endless patience and enormous help in the preparation of this manuscript.

I will always remember George Csanyi-Fritz and his continouos support, if everything had ended we just had a good chat in Hungarian. I have to thank Marlene Cheng for her English writing lessons. I wish I could list all my friends and fellow students who helped me, or had a drink with me, and the ones with whom I regularly played soccer. I cannot help mentioning my 'thinking buddy', Panos.

Finally, I would like to express my gratitude to my mother for her remote moral support throughout my studies in Canada.

I acknowledge the financial support from a University of Victoria Fellowship and the Soros Foundation of Hungary.

"Aki dudás akar lenni,
Pokolra kell annak menni
Ott kell annak megtanulni,
Hogyan kell a dudát fújni."
(Hungarian folk song)

Chapter 1

Introduction

Logic programming and functional programming usually compete with each other as alternatives to declarative programming although they have different potential in problem solving. The two paradigms have different advantages since each provides features not possessed by the other. If a relation-based logic language like Prolog is selected, it might happen that some definitions need to be expressed as functions. A similar situation may arise if a functional language, e.g. Lisp, has been preferred. The forced choice between the two sometimes leads to unnatural modes of programming. An integrated language allowing the definition of both functions and relations would eliminate this inconvenience and lead to an even more powerful tool.

1.1 Amalgamating functional and logic programming

One of the main differences between functional and logic programming languages is the way of treating functions.

Functional languages evaluate their functional expressions and allow func-

tions to be passed as arguments. The application of a function results in a value and that value is computed during the evaluation of an expression. On the other hand, logic programming in its pure form lacks function passing and function evaluating capabilities.

Programming in logic means defining relations. The function symbols of logic programming only act as data structures which can be bound to variables by a mechanism called *unification*. During computation these structures are compared but never evaluated. From this point of view the amalgamation of functional and logic programming (in this thesis referred to as 'integration') is undoubtedly a promising idea.

During the last decade one could witness an intensive effort applied to the integration [12, 3, 4]. The work to combine these two classes of languages resulted in several theoretically interesting results, it made clear their connections, similarities, and differences. Because of the similarities the integration has appeared to be feasible. The most successful systems could bridge the dissimilarities to a certain extent and have enhanced the languages making them more practical and applicable to a wider class of problems.

One approach is to embed a logic language in an existing functional language by means of a suitable interface. Languages like QLOG [18] or LOGLISP [28] succeeded this way as the earliest solutions. This approach, however, ignores and does not resolve the differences of the semantical backgrounds.

It is desirable to achieve integration in a less superficial way. It is possible to define semantical extensions for both of the existing paradigms [26] [23]. Languages such as Babel [24] or Alf [14] extend the functional interpretation towards logic semantics. While solutions like extended unification [19] and

the application of higher-order logic [23] are semantical extensions of the part of the logic programming paradigm that has been realized in practice. These solutions, however, had to face several, often unsolvable difficulties. The obstacles standing in the way of developing usable programming languages are expensive implementation and difficult semantics.

Most of the above mentioned approaches require completely new implementations although both functional and logic programming have well established and efficient implementation techniques.

1.2 Functional programming in logic

In this thesis we discuss an integration idea by introducing function definitions and functional expressions to logic programming. The integration can be achieved by allowing the user to define functions which can be used in the arithmetic expressions of Prolog.

Definitions of functions are given by means of equations. For example, the following definition of the *factorial* function is legal in our integration.

```
fac 0 = 1
fac X = X * fac(X-1) <- X > 0
```

Read the infix '<-' as the function application operation: the left-hand argument is applied as a function to the right-hand argument. These definitions are used in functional expressions given as queries. The value of the expressions can be assigned to logic variables and passed as arguments. The *factorial* function can be used in an application given as a query:

```
?- X is fac 10.
```

The evaluation of expressions is initiated by compiled Prolog queries

```
?- apply(fac,10,X)
```

Expressive examples show the strength and point out the limitations of this solution. During the experiments the efficiency of the approach is studied and the properties of the resulting Prolog code are discussed. Some improvements to the compilation process are suggested. An important benefit of the functional language implementation is that the experience can be used as feedback towards the design of an integrated language. It helps us to determine further properties of the proposed integration and would lead to further design decisions regarding details of the language.

The solution proposed would allow the user to define functions in a Prolog environment. The functions can be passed as parameters. Although our approach is not as flexible in use as the so-called 'lazy' evaluation, it has the advantage that values of functional expressions can be computed in a logic framework. The resulting code can be efficiently executed using conventional implementation techniques, e.g. on a WAM-based Prolog implementation.

Furthermore, translating functional notation into Prolog code and implementing the entire process in Prolog has several advantages compared to other methods. It makes it possible

- to remain in Prolog entirely,
- to maintain the differences of logic and functional variables,
- to use existing tools avoiding the development and implementation of any new machinery required to evaluate functional expressions and relations at the same time. This is especially valuable considering the availability of efficient Prolog implementations.

1.4 Outline of this thesis

Chapter 2 gives a brief review of the basics of functional programming. Chapter 3 introduces our integration proposal and explains its features. Chapter 4 discusses the theoretical basis of the translation, while Chapter 5 describes how the idea is used to compile a functional language to Prolog. Chapter 6 provides some experimental results of our solution after presenting a functional language implementation based on the compilation. Finally Chapter 7 lists our conclusions and suggests future directions for research.

Chapter 2

Lambda calculus and Functional Programming

Properties of functional programming such as easy understandability, high level of abstraction, and an inherently mathematical nature have attracted the attention of both academics and commercial users. Availability of faster implementations has removed the most important argument against its acceptance: its inefficiency. In this chapter we give a short overview of the fundamental principles and ideas behind functional programming.

2.1 The lambda calculus

Let us begin with a brief discussion of the most commonly used theoretical base for functional languages, lambda calculus. For more detailed explanation of lambda calculus the reader should consult [15] or [2].

Lambda calculus, introduced by Church [9], is a mathematical notation of functions. Although its original attempt to provide a foundation for mathematics more or less failed [2], the calculus is useful in studying functions and their applicative behaviour. The formal lambda calculus, a type-free

theory denoted by λ, regards functions as rules in order to express their computational aspects.

Because of its notational simplicity and semantic expressiveness, lambda calculus is used to advantage in expressing the semantics of conventional languages. It is also used as a bridge between high-level functional languages and their low-level implementation [25].

2.1.1 The lambda notation

We can consider lambda calculus as a simple language to express functions. Its most attractive properties, notational simplicity and semantical expressiveness, are due to the fact that it uses only a few syntactic constructs.

Given an infinite set of variables $x_0, x_1, \ldots$, the set of lambda terms, Λ, is defined inductively as follows:

1. **Atom** $x \in \Lambda$,
2. **Abstraction** $M \in \Lambda \Rightarrow (\lambda x\ M) \in \Lambda$,
3. **Application** $M, N \in \Lambda \Rightarrow (M\ N) \in \Lambda$,

where x in the **Atom** and the **Abstraction** is an arbitrary variable.

Application is denoted by juxtaposition. We use the convention that function application is left-associative, and it is customary to omit the parentheses of the lambda term $(M\ N)$ and the outermost parentheses if no confusion arises. In usual models of the λ-theory, Λ contains individual constants (e.g. 1, 2, 3, ..., TRUE) and function constants (e.g. $+, -, *, >$). These constants can be represented by appropriately chosen lambda terms.

The essence of this notation is the lambda abstractor and lambda variables. Functions are expressed by abstraction and the formal parameters of a function are identified via the lambda variables. A term of the form $\lambda x\ B$ represents a function of x. Application is a primitive operation of the theory. In the expression[1] $(\lambda x\ B)N$, the function $\lambda x\ B$ assigns a value to the argument N which can be determined by substituting N for x in B. For example the term $\lambda x\ +\ x\ 2$ defines a function which increments its argument by 2. Thus the application $(\lambda x\ +\ x\ 2)\ 3$ leads to 5.

Via application it is possible to pass functions as arguments. Functions can also be returned as values. Functions taking functions as arguments or returning functions as results are often called *higher order* functions. As an illustration let us consider the function *twice* which applies its first argument to its second argument two times.

$$twice = \lambda f\ \lambda x\ f\ (f\ x)$$

Notice that *twice* takes a function as its first argument, hence it is higher order.

The logical question to ask is what kind of functions are definable as terms. It is shown that, restricted to numerals, exactly the recursive functions are definable. As we will see, although lambda calculus can be used to define recursive functions, the tool and notation of doing so is rather awkward.

2.1.2 Free variables

To discuss the rules of dealing with lambda terms, their equality, and substitution, we have to distinguish between free and bound variables. A variable

[1] In lambda theory the word *term* is used to refer to syntactical constructs. In a more practical context we prefer to use the word *expression*.

x occurs *free* in a lambda term M if x is not in the scope of a lambda abstraction λx, x occurs *bound* otherwise. The scope of the abstraction λx in the term $\lambda x\ B$ is B. If a term has no free variables we say it is *closed*. As an example we can take the term $(\lambda x\ y\ x)$ where y is free, while x occurs bound. This also means that the term is not closed.

The set of free variables $FV(M)$ of an expression M is defined inductively as follows.

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

2.1.3 Substitution

As shown, the value of a function application is obtained by the substitution of the argument for the formal parameter. The result of substituting N for the free occurrences of x in M is defined as follows, where $[N/x]$ denotes the substitution operation.

1. $[N/x]x\ =\ N$,
2. $[N/x]y\ =\ y$ if $x \neq y$,
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 $y \notin FV(N)$.

Special care is needed in the case when $[N/x]\lambda y\ M$ with $x \neq y$ and $y \in FV(N)$. In this situation, known as the *name capturing* problem, substituting N for x in M will cause the free occurrences of y in N to become bound by the λy. To handle this situation we must perform a change of the name of the bound variable y in $\lambda y\ M$.

6. $[N/x]\lambda y\ M = \lambda z\ [N/x]([y/z]M)$ provided that $x \neq y$, $y \in FV(N)$, and z does not occur in M and N.

In order to solve the name capturing problem we can also apply the variable convention: if $M_1, \ldots, M_n$ occur in a certain context, then in these terms all bound variables are chosen to be different from the free variables. If all its variables are different, the expression is said to be in its *standard form*.

2.1.4 Conversion rules

The theory of lambda calculus studies the equivalence of lambda terms based on convertibility. The basic equivalence relation is generated by axioms. These axioms are given as conversion rules.

The renaming we mentioned above is assured by the rule called *α-conversion*. The *change of the bound variable* in a term M is a replacement of a subterm $\lambda x\ N$ of M by $\lambda y\ ([y/x]N)$, where y does not occur in N. α-conversion expresses the equivalence of two terms like $\lambda x\ *\ x\ x$ and $\lambda y\ *\ y\ y$ as one would expect informally. We say that the two terms are *α-convertible*.

A lambda abstraction represents a function. The result of applying the lambda abstraction to an argument is an instance of the body of the lambda abstraction in which free occurrences of the lambda variable in the body are replaced with the argument. The rule expressing this is called *β-conversion*.

$$(\lambda x\ M)\ N = [N/x]M$$

Using the β-conversion rule in an application is the *β-reduction*. If a term M' can be obtained from M by a finite series of β-reductions then we say *M β-reduces to M'*. In order to avoid name capturing during β-reduction,

changes of bound variables may be necessary Any application in the form $(\lambda x\ M)\ N$ is called a *β-redex* and $[N/x]M$ is called *contractum* A term is in *normal form* if it contains no redexes

We can use the *β-conversion* equivalence rule backwards, introducing new lambda abstractions This way, for example, we can alter the term $+\ 3\ 2$ into $(\lambda x\ +\ x\ 2)\ 3$ This operation is called *β-abstraction*

The evaluation of a lambda term means reducing it until its normal form is reached Several questions now arise Does a normal form always exist? When it does, is it unique? How can one reach it?

We should be aware, that in a given term there might be more then one redex to reduce As an illustration let us examine the following problem What should the term $(\lambda x\ 3)((\lambda y\ y\ y)(\lambda y\ y\ y))$ reduce to? This term contains two redexes The function application $(\lambda x\ 3)\ N$ would reduce to 3, no matter what the argument N is On the other hand, attempting to reduce the actual argument $(\lambda y\ y\ y)(\lambda y\ y\ y)$ will lead to an infinite sequence of reductions

The alternatives differ in the order of the reduction If we always choose the leftmost outermost redex first, we perform *normal-order* reduction Simplifying the argument before applying the reduction is referred to as *applicative-order* reduction From the above example, i e from the term $(\lambda y\ y\ y)(\lambda y\ y\ y)$, one can also conclude that not every term has a normal form The first Church-Rosser theorem guarantees, however, that a term has at most one normal form The second Church-Rosser theorem states that normal-order evaluation will always lead to the normal form, if one exists

Besides the general equality-axioms like reflexivity, symmetry, transitivity, and substitutivity, there is one more rule usually included in a formal-

ization of lambda calculus The η-conversion rule expresses the equivalence of lambda abstractions as functions

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

η-conversion satisfies our intuition that two functions are equal if they compute the same result for the same argument

2.1.5 Recursive expressions and the Y combinator

The theory of lambda can be used to express recursive functions Moreover, it is proven by Kleene [2] that exactly the recursive functions are definable in lambda calculus It cannot be accomplished in a straightforward manner because lambda calculus has no facility to name functions, unlike high-level programming languages But it is certainly possible to pass functions into the body of lambda expressions

A typical example would be the *Fibonacci* function

$$\text{fib}(x) = \text{if } x \leq 2 \text{ then } 1 \text{ else } \text{fib}(x-1) + \text{fib}(x-2)$$

The *Fibonacci* function calls itself to compute its value, hence it is recursive

The key to this structure in lambda calculus is the definition of the boolean values *true* and *false* as functions

$$\text{true} = \lambda\text{x}\lambda\text{y}\,\text{x}$$
$$\text{false} = \lambda\text{x}\lambda\text{y}\,\text{y}$$

If-then-else, or IF for short, can also be expressed in lambda calculus

$$\text{IF} = \lambda\text{x}\lambda\text{y}\lambda\text{z}\;\text{x}\;\text{y}\;\text{z}$$

where the first argument returns one of the boolean values. The operator *if-then-else* is usually treated as a constant or built-in function. The boolean values *true* and *false* are also considered to be constants of the notation. Defining $<$ or other conditional operators would be more complicated but they still can be given as lambda expressions [2]. They return either true or false as result. Now we have

$$\text{FIB} = \lambda x.\ \text{IF}\ (\leq\ x\ 2)\ 1\ (+\ (\text{FIB}\ (-\ x\ 1))\ (\text{FIB}\ (-\ x\ 2)))$$

By β-abstraction we can introduce an extra parameter to enable the function to be passed to itself.

$$\text{F} = \lambda f\ \lambda x.\ \text{IF}\ (\leq\ x\ 2)\ 1\ (+\ (f\ (-\ x\ 1))\ (f\ (-\ x\ 2)))$$

So we get

$$\text{FIB} = \text{F FIB}$$

Here FIB is said to be the *fixed point* of F. To compute a fixed point of a function, lambda calculus defines a *fixed-point combinator*[2] to be a term M such that $\forall$ F. (MF) = F(MF). Furthermore, the fixed-point combinator can be expressed as lambda abstraction. One possible definition, called Y, is given by the following term.

$$\text{Y} = \lambda f\ (\lambda x.\ f\ (x\ x))(\lambda x.\ f\ (x\ x))$$

Now we have a non-recursive definition of the *Fibonacci* function!

$$\text{FIB} = \text{Y F}$$

[2]The name *combinator* will be clarified in Section 2.5.2.

We end our discussion with the conclusion that despite the lack of names, recursive functions can be expressed in lambda calculus via the fixed point combinator Y. Following the above train of thought and considering the structure of Y, one may conclude that computing with and implementing the Y combinator does not seem to be a straightforward task. It is certainly not without some peculiarity and 'naughtiness'. Nevertheless, as it is possible to express conditional and recursive structures, lambda calculus becomes a feasible basis for defining functional languages.

2.2 Defining functions using equational notation

Functions, even recursive ones, can be expressed using sets of equations.

The idea was first introduced by Kleene in a 1936 paper "General recursive functions of natural numbers" (referred to in [29]). To demonstrate how equations form a computational formalism let us consider a simple example.

$$f(0) = 0$$

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

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

These three equations together define the function[3] $\lambda x.\ 2 * x$. Notice that f is the main symbol and that g is an auxiliary one. The three equations define one computation by the means of two functions f and g such that the functions call each other *mutually*. The first equation is the base case ensuring termination.

[3] Or $f(x) = 2 * x$

This system of recursion equations has a very limited domain. It computes expressions of the form `0+1+ +1` where `+` associates to the left. Introducing the usual notation of the successor function $s(x)$ for the postfix unary constructor function `+1`, we have

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

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

The equations are universally quantified over their variable, thus the same identifier denotes the same variable throughout an equation. Every variable occurring on the right-hand side of an equation must be included on the left-hand side. In the above example each equation has only one variable, but this is not necessarily the case in every equational formalism.

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

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

The first argument of the application operator ' ' is a term denoting a function. This is in contrast to the value-oriented version, where all terms denote

numbers. Also, here the ' ' is the only evaluable function. Applicative style has the advantage that the constructor functions are easier to recognize.

2.3 Syntactical variants

Lambda calculus can be viewed as a basis for defining programming languages. By the analysis of Turing it follows that in spite of its very simple syntax, the lambda calculus is strong enough to describe all mechanically computable functions [2]. If we consider lambda calculus as a language, lambda abstraction is appropriate to provide scoping and parameter passing. Conditionals and recursive definitions can also be expressed using special lambda expressions. Furthermore, functions can not only be passed as arguments but can be returned as results as well. Even though it would be unusual to write a program in lambda calculus right away, the majority of functional languages are based on lambda calculus to a certain extent.

2.3.1 The ISWIM notation

Several high-level programming languages employ an extended syntax modeled after the language ISWIM by Landin [20]. The notation of ISWIM can be viewed as a syntactic variant of lambda calculus as it has a widely known translation to lambda notation. As a programming language ISWIM offers scoping, parameter-passing mechanisms, conditionals, and recursive definitions.

Let ... in for scoping. The expression of the form

let x=B in E

allows local definitions for both constants and functions. For example,

$$\text{let f(x,y) = x+y in f(2,3) + f(4,5)}$$

is a *let ... in* expression defining a local function , while

$$\text{let pi = 3.14 in 5 * 5 * pi}$$

is a *let ... in* definition with a local constant. Definitions can be nested. We can transform *let ... in* into ordinary lambda

$$\text{let x=B in E} \equiv (\lambda x\ E)B$$

Here the symbol $\equiv$ denotes the equivalence of the two expressions. A multiple *let ... in* expression can be flattened into nested *let ... in* expressions then converted to lambda in a similar manner, leading to nested lambda abstraction. For example,

$$\text{let succ(x) = x + 1 and square(y) = y * y in square(succ(2))}$$

can be transformed to

$$\text{let succ(x) = x + 1 in let square(y) = y * y in square(succ(2))}$$

Conditionals and selection. The operator *if-then-else* is usually treated as a constant or built-in function. The boolean values *true* and *false* are also considered to be constants of the language. For example,

$$\text{let signum(x) = if x = 0 then 0 else if x > 0 then 1 else -1}$$

is a definition with nested conditionals. Notice that here the keyword *let* is used for global definition. By explaining recursive expressions in lambda calculus, it has already been shown (page 13) how *if-then-else* and *truth values* are represented in lambda calculus.

Recursive definitions ISWIM uses a special word *letrec*, similar to *let*, to indicate that the function being defined is recursive Thus its name can occur in the body of the definition For example, the *Fibonacci* function is expressed as

$$\text{letrec fib(x)} = \text{if x} \leq 2 \text{ then 1 else (fib(x-1) + fib(x-2)) in fib(5)}$$

Transforming simple *letrec* to lambda calculus is easy, but requires the use of Y combinator Ordinary lambda calculus has no names, which means referencing is not possible First the recursive definition is translated to *let*

$$\text{letrec x=B in E} \equiv \text{let (x=Y } \lambda\text{x B) in E}$$

then we apply the translation introduced above

$$\text{let (x=Y } \lambda\text{x B) in E} \equiv (\lambda\text{x E) (Y } \lambda\text{x B)}$$

Mutually recursive functions Similar to multiple *let in* definitions, one *letrec in* expression can have more than one function in its definitional part These equations can refer to each other forming *mutually recursive* functions Our example from Section 2 2 in ISWIM notation

$$\text{letrec f(x)} = \text{if x=0 then 0 else g(x)+1, g(x)} = \text{(f(x)+1) in f(5)}$$

2.3.2 Equations

Equations are also often used as the syntax of functional languages The equations

$$\begin{aligned} fib\ 1 &= 1 \\ fib\ 2 &= 1 \\ fib\ (x+2) &= fib(x) + fib(x+1) \end{aligned}$$

give a definition of the *Fibonacci* function if we assume that only positive integers can occur as arguments. The use of equational definitions depends on *patterns*. A pattern is either a *constant*, a *variable*, or a *constructor pattern*, where a *constructor* has some arguments, which are themselves patterns.

Patterns have a crucial role in (equation based) functional programming via a mechanism called *pattern matching*. Pattern matching serves two purposes: first it specifies the form that an argument must take before the corresponding rule can be applied (selection), second it has the affect of decomposing the argument and naming its components — except, of course, when the pattern is a simple data constant — (variable binding) [13]. If patterns in different equations of the same function overlap, the definition is said to be *ambiguous*.

The above definition of the *Fibonacci* function has three rules — one for each possible form that the argument can take. The last equation can also be expressed by the conditional equation

$$fib\ \ x\ \ =\ \ fib(x-2)\ +\ fib(x-1)\ \ if\ x > 2$$

Some languages prefer to use notation close to the equational form. Miranda, a functional language by Turner [34], does not allow lambda expressions to occur on the right-hand side of a definition.

2.3.3 Types

Patterns and pattern matching are closely related to *types*. A *type* states explicitly the domain and the range of a function. There are two ways functional languages handle types. The type of a definition might be automatically derived by the system (as is the case with the language Miranda [34]), or the

user might be required to explicitly specify the type of each function (like Hope [5]). When a function is indifferent to the types of its argument it said to be *polymorphic*. For example, the identity function (λ x x) is *polymorphic* since its argument may be any type.

Through the use of simple types, more complex data types can be formed. Constructor functions have a special role in handling data: they serve to bind data together. Constructors without arguments are called *data constants*. A widely used constructed data type is the *list*, which is defined as consecutive data elements of the same type. A *list* is either empty, denoted by the keyword *nil*, or has two components a *head* and a *tail*. The head is one instance of the data type which forms the list. The tail is another list, which might be empty. Using the constructor name *cons*, the list of the numbers $1, 2, 3$, is represented as

$$cons(1, cons(2, cons(3, nil)))$$

Here 1 is the head of the list, while $cons(2, cons(3, nil))$ is the tail. Another syntactical version using a different constructor is given as

$$[1, 2, 3]$$

The main programming advantage of using a typed language is that it helps the programmer to avoid applying a function to an inappropriate type. Typed languages have built-in *type-checking*, making this possible.

2.4 Evaluating functional expressions

Executing a functional program means evaluating an expression. The value to be computed by a program is obtained by reducing the expression. During

the evaluation of a functional program, one of the most important semantic questions concerns when the parameters are evaluated

2.4.1 Lazy vs eager evaluation

Parameter passing is an important feature of programming languages Lambda calculus provides it naturally by the formal variable of lambda abstraction In functional programming the applicative order reduction of lambda calculus is associated with *strict* semantics, while normal-order reduction leads to *non-strict*, or *lazy* semantics Evaluating an expression lazily means that the arguments of a function are not evaluated until they are needed This is as opposed to the non-lazy or *eager* evaluation, where arguments are evaluated before the function is applied The former is often referred to as *call-by-need*, while the latter is known as *call-by-value* Call-by-value is typically used for parameter passing by imperative languages

The distinction among the various evaluation methods has semantical significance, recalling the reduction orders of lambda calculus While lazy evaluation has the advantages that every expression having a normal form can be reduced to its normal form, and that it allows the definition of infinite data structures, a lazy implementation is generally slower than its eager counterpart On the other hand, strict evaluation does not always terminate, since it does not guarantee finite evaluation (recall the example $(\lambda x\ 3)((\lambda y\ y\ y)(\lambda y\ y\ y))$ from Section 2 1 4, demonstrating the differences of reduction orders)

2.4.2 Binding variables

Another issue related to semantics is the binding of local variables. In functional programming, the value of a variable is determined when the binding expression is *defined*, rather than when the variable is *used*. The latter situation is said to use *dynamic binding* as opposed to *static*, or *proper binding*, where the substitution has to be completed before an expression is passed into an abstraction.

The binding problem typically manifests itself in nested *let ... in* expressions:

let a=2 in let b=a+2 in let a=3 in b*a

If the variable 'a' is not replaced by its value (2) in the definition of 'b' (and thus 'b' is not bound to its value before evaluating the inner subexpression), then the value of the second occurrence of the variable 'a' (3) would be used to calculate the value of 'b' in the subexpression 'b*a', leading to a different result. Static binding is also referred to as proper binding, since often the intended meaning is that 'b' be computed using the outer occurrence of the variable 'a'.

2.5 Implementing functional languages

A functional language can be implemented either by an interpreter or by a compiler. Interpreters usually use lambda calculus, or some variant, as an intermediate code. Compilers generally produce abstract machine code, which requires several transformation steps.

Translating a functional language into lambda calculus or executable code is usually a complicated task. It is typical to take a step-by-step approach

Many functional language implementations compile to an intermediate code which is often regarded as a *'sugared'* version of the lambda notation

From a computational point of view the most important conversion rule is β-reduction The majority of problems arise from the question of how to implement function abstraction efficiently

2.5.1 Environment-based implementation

In a function application, where the function is given as a lambda abstraction, the argument replaces the formal parameter by substitution A *copy-based* implementation creates an instance of the function, actually replacing the variable in the body of the function with its value Instead of performing the substitution, it is possible to keep the variable-argument pair in an *environment* At any given time, the environment contains the actual bindings valid in its scope During the evaluation of an expression, the value of a variable is looked up in the most recent environment One of the earliest environment-based implementations is the SECD machine[4] of Landin The basic implementation of the SECD machine performs eager evaluation With a special delay mechanism, it can be extended to provide lazy features as well

Relying on an environment as a mapping for variable bindings means that the value of an expression depends on its context, if this expression contains free variables or, in other words, it is not closed The presence of free variables is a particular source of danger when a complex expression is returned as value In order to allow (partially evaluated) functions as values we can make an expression with free variables into a constant by closing

[4]SECD stands for Stack, Environment, Code, and Dump the four main components of the SECD machine

$$\begin{aligned} \mathbf{S} &= \lambda f\ \lambda g\ \lambda z\ fz(gz) \\ \mathbf{K} &= \lambda x\ \lambda y\ x \\ \mathbf{I} &= \lambda x\ x \end{aligned}$$

Figure 2 1 The definition of S,K, and I combinators

it with the set of free variables A closed expression could be treated as a constant This composite structure is called a *closure* Returning a closure as a value requires the environment to be copied for the reason that the associated values of the variables are in the environment This is a source of great inefficiency

Because it is so complicated to deal with environments and closures, there is an evident question to ask How could we reach a state where an expression would not contain any free variables?

One possible answer is an implementation based on combinators

2.5.2 Combinators and supercombinators

Eliminating variables from logic is a topic that attracted lively interest for a long time in mathematics Curry and Schönfinkel independently invented combinatory logic to solve this problem [11]

Combinators are closed λ-expressions Every lambda-expression can be transformed into an equivalent expression built only from applications of primitive functions and combinators It can be proved that only two combinators, **S** and **K** are required to generate all the possible expressions For efficiency, a third combinator **I** is included (see figure 2 1)

Combinatory logic was considered only of theoretical interest until Turner introduced optimizations making the scheme more efficient for practical im-

plementation [33] He designed a functional language, SASL, where function definitions are compiled into combinator form in an improved way The resulting expressions are reduced by a machine specially designed to implement the reduction of these three combinators only

Designing such a reduction machine is not too difficult and it can be highly optimized since it only has to handle a fixed set of combinators Compiling expressions to **SKI** combinator form is also easy Besides, the implementation of a reduction machine based on combinators would allow lazy evaluation The penalty for these advantages is high, however Translating the function definitions is time-consuming The prohibitive length of the resulting combinatory expressions is another serious drawback Debugging the code is also difficult As an optimization Turner defined additional combinators

But we do not have to use just a small set of combinators Hughes [16] introduced a special form of combinators in which the lambda abstractions are fairly easy to instantiate The particular property of the construct called *supercombinator* is that no lambda abstraction in the expression contains free variables A lambda abstraction of the form $\lambda x_1\ \lambda x_2 \quad \lambda x_n\ E$ is a supercombinator S, if

1 S has no free variables,

2 any lambda abstraction in E is a supercombinator,

3 $n \geq 0$

In order to reach a supercombinator form it is necessary to abstract out the free variables of a given expression The general process of transforming lambda expressions into supercombinators is known as *lambda-lifting* [17]

As an illustration we can take the following example

$$\lambda f\ \lambda x\ fxx$$

This expression is not a supercombinator because the inner lambda abstraction contains f as a free variable Performing a β-abstraction and an α- conversion, we can close the inner expression

$$\lambda f\ (\lambda g\ \lambda x\ gxx)f$$

Naming the inner supercombinator, we form a constant

$$A = \lambda g\ \lambda x\ gxx$$

so the original expression becomes

$$\lambda f\ A\ f$$

In case of multiple arguments the substitution for supercombinators is performed simultaneously

The evaluation of combinators is done by *reduction* based machines The most efficient ones are based on a technique called *graph reduction*, where the expressions are represented by graphs [25]

Chapter 3

Integration of functional and logic programming

The integration problem can be discussed in a general framework based on a way of handling equality. Introducing functional expressions into logic programming via equality offers a simple integration. The method would allow the user to define and use his own functions in a logic programming environment.

3.1 The integration problem

The difference between logic programming and functional programming can be originated from the role equality plays in the languages of these two paradigms. This difference already presents itself in the definition of equality used by the two representative languages Lisp and Prolog. The desire to close the gap between the two uses of equality lies at the root of most integrations of the two paradigms.

3.1.1 Equality in Lisp and Prolog

To demonstrate the significant difference between functional programming and logic programming in the way of handling equality, let us compare Lisp and Prolog. In this discussion we only consider their pure variants. Lisp, the first declarative language designed for symbolic computation, is based on functions, while pure Prolog works with relations.

In Lisp, programming is based on defining and applying functions. The idea is organized around the concept of *value*. Non-atomic expressions are applications of functions. The value of an expression is computed by applying the value of the operator to the value of the operand. The evaluation proceeds with replacing a subexpression with another, possibly simpler, expression equal in value. This ensures that the value of the whole expression will not change during the process. The definitions introduced by the user are definitions of equality of expressions and are used as such during the evaluation of a program. Although the several syntactical and implementational variations of functional programming are far from the original, value-oriented form of pure Lisp, equality of values is still the heart of the method of functional programming.

This notion of equality is missing from pure logic programming. The power of functional expressions, especially that of higher-order functions, is not provided by simple logic programming implementations. Programming in logic is usually viewed to mean defining relations. Although function symbols occur in terms, they serve as data structures and are never evaluated. The most used logic programming language, Prolog, succeeded by *avoiding* equality. The evaluation of a query to compute an answer proceeds by inference steps. The basic elementary computation of Prolog is unification,

where terms are compared and variables are bound. But unification implements only a limited kind of equivalence, syntactical equality.

It is, of course, possible to express equality in logic as a relation. The implementation of Prolog, however, does not have the potential to fully handle it. The `is` built-in predicate is a form of the equality relation which goes beyond unification. But it is only a partial implementation of equality since not every possible question of equality can be asked and answered via `is`. It extends unification in only one way: a variable can be bound to the value of a certain kind of expression.

3.1.2 Design decisions

It is possible either to view functions as single-valued relations or to consider relations as truth-valued functions. Whichever alternative is selected, the resulting relation-function connection is commonly used as basis for the integration. The choice suggests making the relational language a sublanguage of the functional language or the other way around.

Which should be the sublanguage? This is the very first question to face. We believe that this can not be answered *a priori*. It is certainly hard to reason indisputably that one of the two possible choices is superior to the other. One should choose one way or the other, based on his preferences. There are successful answers from both directions.

Where to interface the two components? It is of course possible and reasonable to include one language in the other. Indeed, this scheme was followed by most of the earlier approaches [10, 30, 21, 32]. The most successful ones embed a logic language in an existing powerful functional language by

means of a suitable interface This is the case in languages such as LOGLISP [28] or QLOG [18] This approach, however, ignores and does not resolve the difference between logic and lambda variables

Extending one of the paradigms towards the other is a commonly used approach If the functional option is chosen, a semantical integration can be based on narrowing [26], a kind of generalized term-rewriting with logic variables Several integrations have used narrowing as the execution model Languages such as Babel [24] or Alf [14] proceeded this way Such a solution requires a totally new implementation This task seems counterproductive given the efficient implementation of Prolog

In the case of approaching semantical integration from the logic side, a process called semantical or extended unification can be used to incorporate equality axioms in the fundamental computation The drawback is that the naive use of equations expressing semantical equality [19] extends the size of the search space enormously Furthermore, it is usually necessary to develop some new machinery to solve unification and value equality at the same time This difficulty can be avoided by means of SLD-resolution with equality axioms as in Cheng and Yukawa's language AP [8]

Taking omega-order predicate logic as a basis instead of first-order logic is another way to incorporate functional programming into logic programming The scheme pioneered by D Miller [23] led to several implementations of the lambda-Prolog language [22] This solution also requires a new mechanism to solve the theoretical and practical problems arising from higher-order unification They seem to lose two of the main attractive features of declarative languages syntactical simplicity and semantic clarity

3.2 Integrating functional expressions into logic

Most of the existing integration approaches — independent whether they select the functional or relational framework — have three problems: they are inefficient in execution, lead to obscure source code and appear to be difficult to implement. We derive our constraints from their weaknesses.

The integration must be theoretically sound so that correctness proofs are possible. A mixed language, in order to be welcomed by potential users, has to be usable, should provide a way of defining both functions and relations, and should allow the user to choose between the two. The conceptual model and the syntax should not be too difficult to understand. The language should provide convenient programming not too far in style from the original and widely accepted fashions of recent functional and logic programming. We are convinced that an acceptable amalgamation should try to maintain the speed and memory space needs of pure Lisp or pure Prolog. From the developers' side we believe in simple yet efficient implementation.

3.2.1 Equations versus lambda calculus

Besides the integration of the two declarative paradigms there is an important integration within functional programming as well, the integration of lambda and equational notation. We do not want to restrict ourselves to only one of them. As a matter of fact, we intend to use both, taking advantage of each.

With its scoping, lambda calculus allows one to make certain variables local. Abstractions can be nested to arbitrary depth, which is useful when creating large expressions. Lambda notation allows the manipulation of func-

tions as 'first-class citizens', so they can be bound to variables, passed as arguments, or returned as results But, as we can remember from what was said earlier, it is awkward when it comes to express recursion

Equations, on the other hand, are an excellent vehicle for defining recursion They are more natural and efficient for expressing recursive definitions than the fixed-point combinator, because they use self-referring names Selection is implicit in the availability of multiple definitions, since there can be more than one equation having the same function symbol on the left-hand side

The power of lambda notation is due to the fact that it does not require functions to be given names On the other hand, in functional programming languages the alternative of naming functions is available These names are used in applicative expressions instead of the full function itself

3.2.2 Embedding functional expressions in Prolog

Our choice is influenced by the preference of logic, therefore, we consider functional programming as a part of logic programming Now, as an immediate consequence we have the logic language Prolog as the frame language Considering the pitfalls of both higher-order and extended unification, we propose a simplistic solution for the integration problem by introducing functional expressions into Prolog

The `is` predicate of Prolog as a particular form of equality can only compute with a limited set of expressions Besides the basic arithmetic and bitwise operations this set usually includes the standard functions of calculus (like trigonometric and logarithmic functions etc) This tool allows one to write simple calculations as an expression rather than a relation For exam-

ple, to assign the value of the expression `(2+3)*5` to the unbound variable Z, one can write

```
Z is (2+3)*5
```

instead of the pure relational form

```
sum(2,3,Z1), times(Z1,5,Z)
```

The `is` predicate is special in the sense that it is usually handled outside of logic, yet it is very much part of a Prolog implementation.

We think this opens an excellent opportunity for an extension: why not include *all* functional expressions of a functional language? We restrict the functional part to the second argument of `is`. The introduction of functions into Prolog would in principle allow definitions of equality. This equality can only be used in a restricted way, however, on the right-hand side of `is` predicate.

In this integration approach the functional part is interfaced into the logic part in a well-controlled manner. We call our approach Lambda-Equational Logic Programming or LELP for short. The name was first used in [6] and a different version of a language, λELP was defined. The integration combines three components (see Figure 3.1):

1. Pure Prolog with the usual extensions like `cut` (or !), `not` etc.

2. Conditional equations can be used in the program to define functions. These definitions are *global* and deterministic. Consider for example the following function:

```
list L 0 = L
list L N = list [N|L] (N-1)  - N>0
```

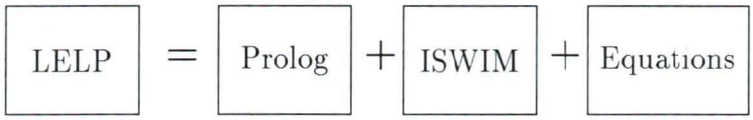

Figure 3.1: Model of the Integration Proposal

The function *list* takes a list and a non-negative integer and appends the first N natural numbers to the beginning of the input list. For example `list [8,6] 2 = [1,2,8,6]`. If applied to the empty list, the function generates the list of the first N natural numbers in ascending order.

3. Functional expressions — possible expressed by ISWIM syntax — can occur on the *right-hand side* of an extended version of `is`. It is used just like regular `is`. It binds the value of the expression occurring as second argument to an unbound variable given as first argument. For example, one might write the following query

```
?- X is (\Y.Y+3) . 5.
```

and get the answer as `X = 8`, where '\' denotes lambda abstraction and '.' denotes application.

3.2.3 Advantages of this approach

This proposal has several attractive characteristics:

1. We don't have to worry about the effect of unification and backtracking in the functional context, since `is`, cannot be backtracked over.

2 The functional expressions are encapsulated in the right-hand side of `is`, so we don't have to face the question how higher-order functions, scoping, and other functional programming features would behave in a logic environment They are detached from the Prolog components

3 Separating the two different kind of components this way leads to controllable interfacing We can keep a tight rein on the communication between them

4 The integration uses well-known functional and logic notation and elements only, so it is easy to understand, handle, and program with

3.3 Implementation issues

How to evaluate the class of programs outlined by the above implementation proposal?

For the evaluation of the Prolog part WAM [1] is a serious candidate But how can the evaluation of the functional parts be implemented? Figure 3 2 shows the question of implementation We consider two main alternatives

1 To evaluate ISWIM a machine like SECD is an acceptable choice A possible way to handle equations is to use them as rewrite rules as in a term-rewriting system (TRS) Three difficulties stand in the way of this solution First, the conditional equations might not form a set of confluent and terminating rewrite rules Second, the connection between the SECD machine and the TRS needs to be clarified But most importantly, the implementation of such a mixed machine (Figure 3 3) is still an open research problem

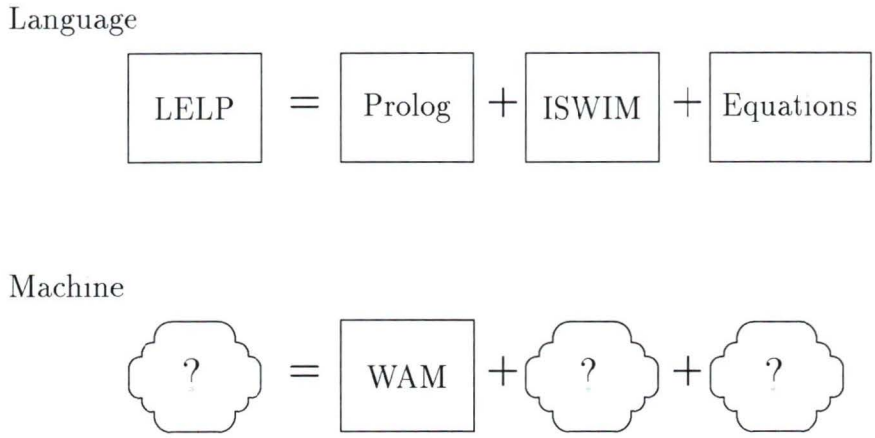

Figure 3.2: Question of the implementation

2. The other alternative is to translate the functional expressions into relations and use WAM for the evaluation of them as well. The Prolog part remains unchanged.

   The translation is done in two steps as it is shown in Figure 3.4. First lambda expressions together with ISWIM expressions are translated into equations. Then all the equations are translated into logic clauses. The clauses form an ordinary Prolog program, and queries containing functional expressions can be evaluated using conventional Prolog implementations.

Most other approaches to integration require completely new implementations although both functional and logic programming have well established, efficient implementation techniques. In the second of the above alternatives, Prolog is not affected, so WAM can be used as its implementation. Since functions are translated to clauses, WAM can be used for their evaluation as

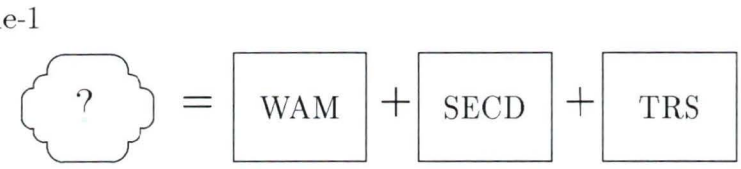

Figure 3.3 Implementation model with three machine components

well.

The second alternative promises an easier solution as for the developers' side. The rest of the thesis is devoted to the investigation of a possible implementation of this integration based on the proposed translation.

Our starting point is that the Prolog part remains the same, so we only have to deal with the functional components: the equations and lambda expressions.

The investigation is done in three increments. First we summarize the theoretical background for the two-step translation. Chapter 4 will show how to translate individual expressions to logic clauses. The functional components of the integration can form a functional language. There are function definitions and expressions to evaluate. As the second step we define a compilation scheme from a functional language to logic programs and queries. Using the two-step translation Chapter 5 will analyse the problems arising when we have more than one definition (or equation) at a time and more complex expressions with possible ISWIM syntax (e.g. nested *letrec ... in* expressions, pattern matching etc.). Chapter 6 — as the last stage in our investigation — will consider the implementation of a functional language based on the compilation defined in Chapter 5.

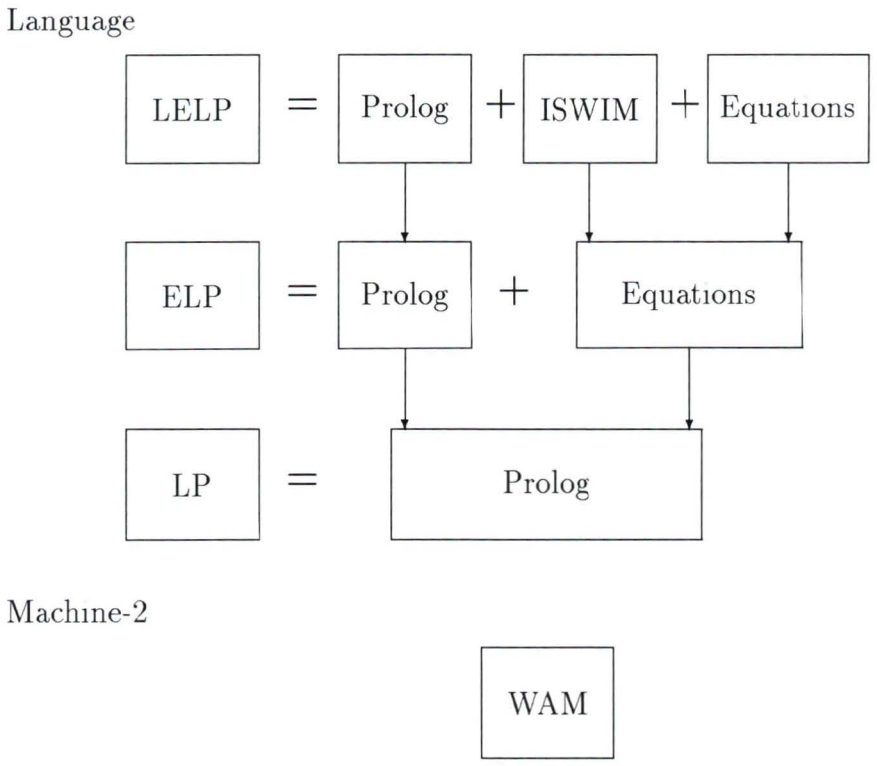

Figure 3 4 An implementation based on translation

As the conclusion of the discussion we will see whether the translation method is a promising and suitable idea for implementing the LELP integration We will also be able to determine what language features this implementation would allow

Chapter 4

Translating functions to relations

The Prolog implementation of the functional part of LELP is based on a two-step translation. First the lambda expressions are transformed into equations. In the second step these equations, along with other definitions originally given as equations, are translated to logic clauses. This method has the advantage that both translations can easily be implemented in Prolog.

4.1 The relation of equational and lambda notation

Both lambda expressions and equations can be used to define functions. The notation of recursion equations provides the function name and its arguments on the left hand side of the equation, and the definition itself is given on the right hand side. The supercombinator form of lambda calculus uses named lambda expressions, where the right-hand side of the equation is a closed lambda abstraction. The lambda-lifting procedure used for the supercombinator technique suggests that the two notations for defining functions

are strongly related. Indeed, each lambda abstraction can be lifted into an individual equation.

There are functional languages which allow both alternatives, or even a transitional form between the two. For example, in Scheme[1] one may have three ways to define the *twice* function.

```
(DEFINE TWICE (LAMBDA F (LAMBDA X (F (F X)))))
(DEFINE (TWICE F) (LAMBDA X (F (F X))))
(DEFINE (TWICE F X) (F (F X)))
```

The first choice is similar to a named lambda expression

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

the last looks like an equation

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

while the second is a mixture of the two alternatives

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

Comparison of these choices suggests that the form $f = \lambda x.\ B$ can be derived from $f\ x = B$ and vice versa.

Consider first the lambda abstraction form

$$f = \lambda x.\ B.$$

Assuming that x is the only free variable in B we can apply both sides to x. The left-hand side becomes $f\ x$ while the right-hand side is reduced to B by β-reduction. Hence we conclude:

[1]Scheme is a functional language designed by Steele and Sussman [31].

$$f\ x = B$$

Conversely, let us assume that

$$f\ x = B$$

such that x occurs free in B It is possible to apply lambda abstraction to both sides

$$\lambda x\ (f\ x) = \lambda x\ B$$

η-reducing the left-hand side gives

$$f = \lambda x\ B$$

which is the desired equality Hence we conclude that the abstraction and application operators are interchangeable (Figure 4 1)

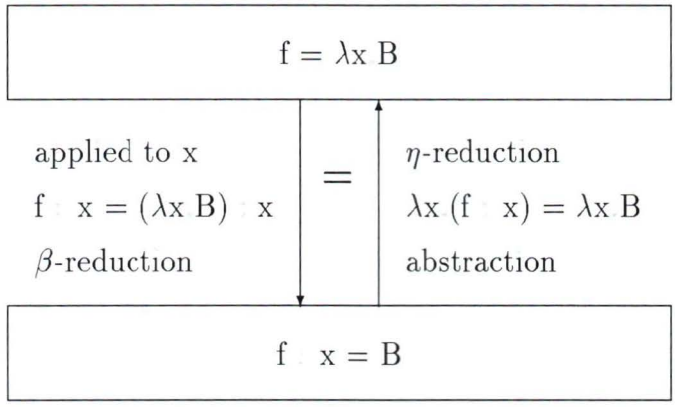

Figure 4 1 Interchangeability of abstraction and application

4.2 Translating lambda abstraction into equational form

We can give a formal definition of the transformation from lambda expressions to equations. There are still two problems we have to solve.

The interchangeability is sufficient only if there are no further abstractions in B. However, the process can be repeated as many times as is necessary to remove all occurrences of λ from the original expression. Abstractions are replaced by their introduced unique names. This means that nested lambda abstractions are translated into a set of equations, each lambda abstraction leading to one equation.

The other problem is to decide what should happen to the free variables occurring in the body of the lambda abstraction. The obvious solution is to provide them as a closure.

The approach, called lambda elimination, was defined by M. Cheng and first appeared in [6]. Given a lambda expression t, its translation to equations denoted by t^λ is defined inductively as follows. Here ' ' denotes application, as used in [6].

1. $t^\lambda = t$ if t is a variable,
2. $t^\lambda = t_1^\lambda \ t_2^\lambda$ if $t = t_1 \ t_2$,
3. $t^\lambda = f(y_1, \ldots, y_m)$ if $t = \lambda x \ t_1$, $y_1, \ldots, y_m$ are the free variables of t, and f is a new function symbol. The equation $f(y_1, \ldots, y_m) \ x = t_1^\lambda$ is added to the set of equations.

Notice, that the original lambda expression is replaced with another expression containing no lambda abstraction, and a (possibly empty) set of

equations is generated The resulting expression is either a simple constant, a term representing a closure, or an applicative expression composed from these three elements

This method is certainly not without some similarity to the supercombinator forming procedure Johnsson's lambda lifting algorithm[2] [17] was devised to lift local function definitions using lambda expressions possible containing free variables to global function definitions using combinators His approach closes open subexpressions with additional lambda abstraction(s) A new abstraction is introduced for every free variable Instead of abstracting out subexpressions containing free variables, our approach closes the subexpression with a closure The process goes inwards from the outermost expression Every lambda abstraction is turned into an equation regardless of whether it contains a free variable or not In the discussion, [6] shows that the latter approach is more efficient and requires fewer operations depending on the number of function definitions Even higher performance can be gained by carefully designing a language combining equational and lambda expressions together, taking advantage of both notations

In [6] several other useful results are proven as well, including a proof that the compiled equations can be used as rewrite rules to simulate β-reduction It is shown that the translation can be done both ways The reverse translation, called lambda introduction, can be used later to transform partially evaluated functions into a more readable form Thus, unlike most functional languages, it allows a function to be returned as a lambda abstraction

As an illustration, we will show how to eliminate lambda abstractions

[2]The technique was introduced for lazy functional languages but can easily be generalized for strict semantics as well

from the expression $e = (\lambda f\ \lambda x\ \lambda y\ f\ x\ y)$ There are no free variables in e, so a constant function name like g_1 is appropriate to replace it The transformation requires the following steps

1 $e^\lambda = g_1$ with a new equation $g_1\ f = e_1^\lambda$
where $e_1 = (\lambda x\ \lambda y\ f\ x\ y)$,

2 $e_1^\lambda = g_2(f)$ with a new equation $g_2(f)\ x = e_2^\lambda$
where $e_2 = (\lambda y\ f\ x\ y)$,

3 $e_2^\lambda = g_3(f,x)$ with a new equation $g_3(f,x)\ y = e_4^\lambda$
where $e_4 = f\ x\ y$,

4 e_4 contains only applications and variables so it remains the same, thus
$e_4^\lambda = e_4$

Eliminating all lambda abstractions from e thus results in g_1, and the set of equations

$$\{\ g_1\ f = g_2(f),\ g_2(f)\ x = g_3(f,x),\ g_3(f,x)\ y = f\ x\ y\ \}$$

The resulting equations have two possible interpretations in logic With respect to logic syntax, the lambda variables are replaced by logic variables Different lambda variables has to be replaced by different logic variables

- Each equation is considered to be an an atomic formula of logic This interpretation takes the viewpoint that a function is a set of equations The set of equations as unconditional clauses form a logic program In this sense, for example, the *twice* function is defined in logic as follows

$$twice = \{\texttt{tw\_1 F=tw\_2(F), tw\_2(F) X=F (F X)}\}$$

The occurrences of the variable F in different equations denote different logic variables.

- Another aspect of the relationship between the lambda expression and the equations can be emphasized by treating the equations as conditions in a logic clause, such as

```
\F \X F (F X) = tw_1 - tw_1 F = tw_2(F),
                       tw_2(F) X = F (F X)
```

Here, since the equations are appearing in the same clause, the same variable name stands for the same variable.

We do not see one alternative as inherently superior to the other. They have different roles in our integration approach. The first alternative can be used during execution, while the second one helps in the translation process.

Automating the translation in Prolog is not too difficult. It consists of disassembling expressions, finding their free variables, and including them as arguments to an introduced functor. In Appendix B, [27] provides a partial implementation of the process of lambda-elimination in Prolog. The work also explains a method showing how SLD-derivation can perform the reverse translation.

The equations resulting from lambda elimination were used as rewrite rules in a logic-based term rewriting system. [36] and [7] discuss the practicability and implementation of such a term rewriting system. In these papers a logic program was presented, where SLD-derivation 'mimics' a rewriting algorithm. The equations are added as clauses to the logic program. This

technique, called 'rewriting by resolution', enables evaluation of functional expressions in logic programming. Major difficulties include detecting unsuccessful rewrites and avoiding infinite branches of resolution. [27] reports several improvements to the technique.

4.3 The relationalization axiom

Rewriting by resolution might be one way to incorporate functional programming in logic programming. One major disadvantage of this approach is its lack of efficiency. Any rewrite-based system spends a large part of its time searching for subexpressions to rewrite.

[27] presents another approach, where the complex expressions are flattened into subcomputations. Knowing what the simple equations are, the process can be taken one step further. It is possible to compile the equations into Prolog code that will compute their value when called. This procedure is utilized by a process called *relationalization* [35].

The relationalization axiom for a function f establishes a connection between the value of the function and its arguments and a relation *frel* of the same arguments and value.

$$\forall x_1, \ldots, x_n, v \quad f(x_1, \ldots, x_n) = v \leftrightarrow frel(x_1, \ldots, x_n, v)$$

Note, that the axiom is formalized for *value-oriented style*. For *applicative style* the axiom can be expressed as follows.

$$\forall f, x, v \quad f\ x = v \leftrightarrow apply(f, x, v)$$

Also, $apply/3$ is a general relation, since the only evaluable function is the application ' '. Each function is applied to one argument at a time, which

fits our lifted form of equations

The definition of how to compute v, the value of f applied to an argument x, is given by the accompanying equations Using the equations, the result can be computed by rewriting But how can the corresponding `apply/3` relation be defined? The latter version of the relationalization axiom can be used to translate equations expressing functions into relations Besides the axiom itself, the general equality axioms, (i e symmetry, reflexivity, transitivity, and substitutivity) are needed to form a chain of proof which leads from equations to relations

Let us take our favorite example, the twice function Its second equation in logic is $tw_2(F)\ X = F\ (F\ X)$ The logical chain

$$\begin{aligned} apply(tw_2(F), X, V) &\leftarrow tw_2(F)\ X = V \leftarrow tw_2(F)\ X = U,\ U = V \\ &\leftarrow F\ (F\ X) = V \leftarrow F\ X = U,\ F\ U = V \\ &\leftarrow apply(F, X, U),\ apply(F, U, V) \end{aligned}$$

proves the logic program clause

$$apply(tw_2(F), X, V) \leftarrow apply(F, X, U),\ apply(F, U, V)$$

The implications are justified, respectively, by the relationalization axiom, transitivity, the equation, substitutivity, and relationalization again For the last implication, the relationalization axiom was applied in the opposite direction to the very first step The result of the implication chain is a logic clause It is the logic definition of the given function reflecting the applicative style

Considering each link as a goal statement in the sense of logic programming, this chain can be further formalized This way the entire chain can

$\leftarrow apply(tw_2(F), X, V)$

$\leftarrow tw_2(F)\ X = V$

$\leftarrow tw_2(F)\ X = V_1,\ V_1 = V$

$\leftarrow F\ (F\ X) = V$

$\leftarrow F\ X = V_2,\ F\ V_2 = V$

$\leftarrow apply(F, X, V_2),\ F\ V_2 = V$

$\leftarrow apply(F, X, V_2),\ apply(F, V_2, V)$

$apply(X, Y, V) \leftarrow X\ Y = V$

$X = V \leftarrow X = Y,\ Y = V$

$tw_2(F)\ X = F\ (F\ X)$

$X\ Y_1 = V \leftarrow Y_1 = Y_2,\ X\ Y_2 = V$

$X\ Y = V \leftarrow apply(X, Y, V)$

$X\ Y = V \leftarrow apply(X, Y, V)$

Figure 4.2 Deriving clauses by SLD-resolution

be generated as an SLD-derivation. It is necessary to be precise about the form of the axioms used: they are the input clauses of the SLD-derivation, as shown in Figure 4.2 ([27]).

In [27] the translation from equations to relationalized clauses is performed by a meta-interpreter that constructs an SLD-derivation giving the desired clause as a *conditional answer*. In this way the resulting clause is obtained directly from its *proof*.

4.4 Related work

In his paper [37], Warren discussed possible 'higher-order' extensions to Prolog.

Among other ideas, he examined the possibility of referring to a predicate without having to give it a name. This can be achieved, for example, by using

lambda notation in the description of a predicate, such that the body of the lambda abstraction would be a Prolog goal. As an illustration, the following clauses and query were provided:

```
have_property([],P)
have_property([X|L],P)  - P(X), have_property(L,P)
?- have_property([0,1], lambda(X) square(X,X) )
```

meaning the question whether the elements of the list in the first argument (in the example 0,1) has the property given as the body of the lambda expression (in this case of being equal to their squares). The example query would succeed, of course.

Warren regarded this extension as a simple syntactic variant of Prolog. In order to support this claim, a mapping was provided showing how to translate this kind of 'higher-order' objects into first-order predicate logic. A lambda expression

```
lambda(X1,  ,Xn) E
```

is replaced by a unique identifier, say `ph1`, and an `apply/n+1` clause is formed:

```
apply(ph1,X1,  ,Xn)  - E
```

The above query rewrites to

```
?- have_property([0,1], is_its_own_square)
```

with a new clause

```
apply(is_its_own_square,X)  - square(X,X)
```

The clauses for `have_property/2` are also redefined as

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

Warren concluded that, at least for handling functions as first order objects, this extension would not add anything to the real power of logic languages. Moreover, Warren was convinced that his variant better served the program-writing philosophy of Prolog. He also argued for the efficiency of his translation with early (structure-sharing) Prolog implementations.

Although the form of `apply/3` clauses is introduced in Warren's paper, the principles behind the idea are not discussed. Warren showed some examples, but was not concerned about the correctness of the translation. He did not present a general method. The formalization of the idea needed further work [7]. Our relationalization technique can be considered as a formalized version of Warren's idea.

Chapter 5

Compiling a functional language to Prolog

The method of applying the two-step translation as it is described in the previous chapter to implement LELP needs further improvement. Evaluating functional expressions in a language environment requires more than treating simple single expressions. Further steps are necessary to improve the method before it becomes a practical technique and meets our design goal. We can show the feasibility of this particular approach to implement the functional subset of the integration by designing a scheme to compile a functional language.

We take an existing functional language, FPL. Details of the FPL system, its syntax, and the new implementation will be given in the next chapter. In this chapter we consider the steps necessary to extend the translation method to become an implementation technique. The two-step translation is integrated into a one-step compilation to Prolog code. The advantages are easier and more controllable code generation and easy accommodation to different surface syntax. The compilation is more convenient in handling

global and local definitions given by the user and promises faster translation

5.1 A functional language in Prolog

The language FPL and its syntax was designed by Dr M H M Cheng of University of Victoria The original implementation is based on the SECD machine emulated in Prolog We use our approach to replace the SECD implementation

FPL is a typed functional programming language supporting lazy evaluation[1], higher-order functions, universal polymorphism and pattern matching The notation of FPL is modelled after ISWIM, it is a 'sugared', typed lambda calculus

5.1.1 Implementation strategy

When the system is started it has to include the Prolog code of built-in functions A program written in FPL is evaluated by an interpreter The value of an expression is obtained in two steps Using the current context which defines the meaning of built-in and predefined functions and constants, every expression is compiled to a logic variable, a goal, and a set of clauses as Prolog program The context is also provided in its compiled Prolog form The code of a definition is added to the Prolog program of all the definitions defined up to that point In case of an application the Prolog interpreter is invoked to evaluate the query compiled along with the code of local definitions, if any The value of the expression is returned as the binding of the given logic variable after the evaluation of the goal in the presence of

[1]The original implementation of FPL is lazy because of the delay mechanism of the underlying SECD machine Our approach can not support the lazy features

the compiled Prolog program and the definitions of the current context The clauses compiled from global definitions can be saved into a file and used like any other Prolog program Also, the user has a way to load in the code of predefined functions compiled and saved already during a former session

5.1.2 Extending the translation towards a language

Generating unique names Recalling the lambda elimination from Section 4 2, every function is replaced by a newly introduced unique name Function definitions with multiple arguments has one name generated for each argument These names contain the closure of free variables for expressions that were not closed Using lambda elimination in the context of a functional language the situation is a bit more complicated In a program we have more than one expression to translate, besides, function definitions might be expressed by more than one equation We have to deal with two kinds of variables lambda variables and variables denoting functions or constants The last ones are not free, so they will not occur in the closure Lambda variables only occur in the closure if they are free in the subexpression to be translated

When there is more than one function at a time, generating unique names for newly introduced equations needs a little bit of attention If we are to use a general name, say g, we have to provide different indices for different functions and we also have to differentiate equations resulting from nested lambda abstractions within the same function One index can be used to keep track of the number of functions translated, and another index can separate multiple lambda abstractions within one definition For example, if we have

the two functions *twice* and *compose*

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

$$\text{compose} = \lambda x\ \lambda y\ \lambda z\ x\ (y\ z)$$

the result of the translation would be

```
g_1_1   F = g_1_2(F)
g_1_2(F)   X = F   (F X)
g_2_1   X = g_2_2(X)
g_2_2(X)   Y = g_2_3(X,Y)
g_2_3(X,Y)   Z = X (Y Z)
```

We call the main generated name, e g `g_1_1`, a *combinator* As the result of the translations *twice* is replaced by `g_1_1`, while *compose* got the combinator name `g_2_1` If the lambda expression has a given name, we can replace the general identifier `g` and the first index with that name in order to generate a unique name So we get `twice_1` and `compose_1` as combinators

Handling the current context We differentiate two parts of the context environment and declaration The *environment* is the context for bound variables, the *declaration* is the context for all definitions A declaration (or an environment) is just a mapping from identifiers (or variables) to values Each bound variable which has been defined as a function has its combinator associated with it The mapping of variables to combinators is kept in the environment, while the mapping from combinators to functions is provided in the declaration

The environment is used during the compilation of an expression, while the declaration is needed during the evaluation of the compiled code From

implementation point of view the environment is a list of variable-value and variable-combinator pairs. The declaration is given as a Prolog program which contains the compiled code of built-in and predefined functions. It is a set of `apply/3` clauses. As an illustration, consider the declaration of the built-in functions `+` (addition) and `/` (division)

```
D0= { apply(add,X,add(X))
      apply(add(X),Y,Z)  - Z is X + Y
      apply(div,X,div(X))
      apply(div(X),0,error('division by zero'))  - !
      apply(div(X),Y,Z)  - Z is X / Y
    }
```

The functions are represented in their curried form. `Add` and `div` are just our combinator names for `+` and `/` respectively. They are stored in the environment as pairs `(+,add),(/,div)`.

The compiled code. In a given context where variables are given in an environment *Env*, any functional expression *Exp* is compiled into a logic variable `V`, a goal `G`, and a set of clauses as Prolog program `P`. It is expressed by the following compilation relation:

$$\text{CodeOf}\ (Exp,Env) \mapsto (\ \texttt{V}\ ,\ \texttt{G}\ ,\ \texttt{P})$$

`V` might be either bound or unbound. `G` might be empty, represented by the Prolog keyword `true`, otherwise `V` usually occurs in `G`. `P` might be empty as well. The declaration of built-in and predefined functions is a Prolog program, `D`.

An expression translated to Prolog (and to be evaluated as a Prolog program) has to have a way to return its value. We have a query and it is important for the logic variable carrying the *final* result to be identified. To understand the need for distinguishing this compiled logic variable (for which we usually refer to as `V`), consider the following example where the functions *twice* (*tw*) and *successor* (*succ*) have been defined already.

$$tw\ \ tw\ \ tw\ \ succ\ \ 1$$

The Prolog code of this application as it follows from relationalization is

```
?- apply(tw,tw,Z1),apply(Z1,tw,Z2),
   apply(Z2,succ,Z3),apply(Z3,1,Z4)
```

Running this query on a Prolog interpreter such that the Prolog code of the *twice* and *successor* functions is given, the following answer is produced.

```
Z1 = tw_2(tw_1)
Z2 = tw_2(tw_2(tw_1))
Z3 = tw_2(tw_2(tw_2(tw_2(succ_1))))
Z4 = 17
```

Which variable has the value of the application? In this case the variable `Z4` has the final value of the original expression, so it has to be distinguished from the other, auxiliary, logic variables.

Obtaining the value of an expression. If *Exp* is an application, then the goal `G` is evaluated as a Prolog query using the Prolog representation of the definitions in `DUP`. The binding of the logic variable `V` is returned as the value of *Exp*. The evaluation can be expressed by the following relation.

$$\text{ValueOf} \quad (\texttt{V},\texttt{G},\texttt{P},\texttt{D}) \mapsto \texttt{V}\theta$$
$$\text{if } \texttt{D} \cup \texttt{P} \models \forall \texttt{G}\theta$$

If *Exp* is a definition, then the logic variable is bound to the combinator name of the new definition, which is the value of *Exp*. The goal G is empty. The current declaration D is updated with the resulting program D'=D∪P. The current environment is updated as well. For each function definition a pair is added formed by the name of the definition as variable and the unique combinator. The combinator as a newly introduced name will occur in the clauses of the program P representing the Prolog version of the definition. For each constant definition a pair is added formed by the variable and the value of the constant. This value might be computed in case the body of the constant definition is an application.

Global function definitions will be part of the declaration, while local function definitions — appearing in *let ... in* and *letrec ... in* expressions of applications — only kept in the current declaration till the value of that applicational expression is evaluated.

5.2 The compilation scheme

The rules of the compilation scheme are practical reconstructions of the lambda elimination method combined with the proof method for relationalization. The rules describe how expressions of the functional language are compiled to the logic components V,G,P using the environment.

The most basic construct in an expression is the application denoted by the application operator ' '. As we can remember from Section 4.3, the `apply/3` predicate is the logic counterpart of the application operator ' '.

As an overall simplification of the compilation process one might say, that each ' ' operator is turned into an `apply/3` predicate. A definition is compiled into a clause or clauses. In case of ' ' occurring on the left-hand side of an equation, the resulting `apply/3` predicate will be the head of a, possibly unconditional, clause. `Apply/3` predicates compiled from ' ' operators occurring in the body of a definition or in an application will form goals. Goal from a definition will be the body of the clause which head was just defined, goals from applications are treated as queries. The compiled clauses form a Prolog program, which is the Prolog representation of the definitions.

5.2.1 Compiling the 'twice' function: an example

To demonstrate how the compilation works, Figure 5.1 presents the compilation of the *twice* function `tw f x=f (f x)`. It is assumed, that at the beginning there are no predefined functions in the environment. After the first step the environment is not updated with the combinator of `twice`, since *twice* is a non-recursive function. (The definition of *twice* consists of only one equation, also, it is easy to check, whether the body contains the name *twice* as a free variable.)

The compilation of the application `f (f x)` is split into two parts: the compilation of the operator `f`, and the compilation of the operand `(f x)`. The line after the merge is the result of the compilation. As the result of the whole compilation `tw` is returned as value and `P` is added to the declaration `D`. Finally, the definitional part of the environment is now updated with the identifier-combinator pair so the name `tw` can be used by later applications.

The compilation is defined for simple expressions, equations, and lambda

```
Exp                 Env         V          G           P
-------------------------------------------------------------
tw f x=f (f x)      []          V=V1       G=G1        P
tw f x=f (f x)      []          V1=tw_1    G1=true     P1
tw_2(F) x=f (f x)   [F/f]       V2=tw_2(F) G2=true     P2
f (f x)             [X/x,F/f]   V3         G3          P3=PdUPr
split
operand(opd)
f x                 [X/x,F/f]   V_opd=V4   G_opd       Pd={}
operator(opr)
f                   [X/x,F/f]   V_opr=F    G_opr=true  Pr={}
merge
-                   [tw_1/tw]   V=tw_1     G=true      P=P1UP2
-------------------------------------------------------------
where
P1 = {apply(tw_1,F,V2) -G2 }
P2 = {apply(tw_2(F),X,V3) -G3 }
G3 = G_opd, G_opr, apply(V_opr,V_opd,V3)
G_opd = apply(F,X,V4)
P = {apply(tw,F,tw_2(F))
     apply(tw_2(F),X,V3) - apply(F,X,V4), apply(F,V4,V3) }
```

Figure 5.1 Compiling the *twice* function

expressions including ISWIM.

5.2.2 Compiling simple expressions

Constants Numbers and other primitive data elements are compiled to themselves. The compiled logic variable will be bound to this value. The compiled goal is `true` and the set of compiled clauses is empty.

Identifiers The value of both built-in and user defined identifiers as variables is looked up in the current environment. The goal is `true`, and a set of clauses is empty. The value might be a combinator in case of a function.

Arguments of function definitions The formal parameter of a function definition can be either a pattern or a variable. Constant patterns are compiled to themselves. Constructed patterns are compiled as types (discussed later), but they might contain variables. Variables (even in patterns) are replaced with a newly introduced unique logic variable, and pairs formed by the functional variables and the corresponding logic variables are added to the environment. The newly introduced logic variable will replace its functional pair throughout the body of the function definition whenever the argument as a variable is looked up in the environment. The compilation of the *twice* function in Figure 5.1 demonstrates the replacement of functional variables as arguments by logic variables.

Applications Both the operator and the operand is compiled and the resulting clauses are merged, if there is any. The goal is formed by the conjunction of three components: the goal of the operand precedes the goal of the operator, and a new `apply/3` predicate as goal is conjuncted to the end of them. The arguments of the additional `apply/3` predicate are the following. The first argument is the compiled logic variable of the operator, the second argument is the compiled logic variable of the operand, while the third argument is the compiled logic variable variable of the whole application. The order of the `apply` clauses is the reason for our FPL implementation having strict semantics. If the operator or the operand is a function name, its logic variable is bound to the corresponding combinator.

For example the goal resulting from the application `+ 1` is

```
G5= true, true, apply(add1,V5)
```

since the goal of both the operator and the operand is empty. No clauses are

compiled and `V5` is the logic variable. It is not required to write out `true` for the empty goals.

The expression `3+2` compiles to

```
G6= apply(add,3,V61), apply(V61,2,V6)
```

since its applicative form is $+\ 3\ 2$. Here the value of `+` is `add`, and the numbers remain the same. Each of the three components has an empty goal and an empty program. `V61` is the value of the first application. `V6` is the compiled logic variable of the whole expression and will contain the value of the expression after the evaluation of the goal.

Obviously, this latter example can be compiled more efficiently as we will see later. We will also explain why it is necessary and useful to keep the (curried) format presented here.

5.2.3 Compiling equations

Equations can only occur at the global level, so they have no free variables. Otherwise equations are compiled as it follows from the proof method for relationalization.

Single equations. In case of a constant definition the right-hand side of the equation is compiled and the compiled code is evaluated. The resulting value is added to the current environment paired with the constant symbol as a new global variable.

An equation as a function definition compiles to at least one clause. The number of clauses depends on the number of arguments and the structure of the expression in the body of the definition.

First it is necessary to derive the curried version of the definition A function definition with multiple arguments is flattened out into as many equations as there are arguments Flattening out the equations means that, except for the very first equation, free variables can occur in the body of the equation For each additional equation a unique combinator name is introduced containing the closure of free variables in the similar manner as it is done during lambda elimination

Then the normal relationalization applies Equations now having one argument are compiled to an `apply/3` clause, where the body of the clause is the goal compiled from the right-hand side of the equation The first argument is the combinator name occurring as the function symbol of the definition The second argument is the logic counterpart of the argument of function, and the third argument is the compiled logic variable resulting from the compilation of the right-hand side of the equation The resulting clause is merged with the program compiled from the right-hand side of the equation Conditions of equations are replaced by the logic predicate denoting the same boolean operator, e g `x>1` is replaced by `X>1` The boolean operators 'or' and 'and' have their natural counterpart in a logic clause, ';' and ',' respectively 'Not' is compiled as an operator, which returns *false* if applied to *true* and vice versa

The compilation of the *twice* function in Figure 5 1 shows the process for two arguments

Multiple equations Multiple equations defining one function[2] are simply compiled as if they were single equations. It is important to note, however, that they need only one combinator to be stored in the environment. Another important thing to mention is that functions are deterministic, while predicates are not. In case of ambiguous definitions it is necessary to cut off the remaining choices, once a clause was successfully applied. The definition of the division operator in the declaration `DO` on page 56 illustrates the use of cut.

Multiple equations defining mutually recursive functions require the the participating function names to be collected and be put into the environment before the compilation of the individual equations.

5.2.4 Compiling lambda expressions

Lambda expressions can occur both in applications and in definitions. Lambda abstractions without a name will be given a unique combinator name and then they are compiled as any equation. If a lambda abstraction occurs in a larger expression as subexpression, it will be replaced by the new combinator name. The combinator will include the free variables occurring in the abstraction. Identifiers denoting global or local definitions are not considered to be free, since they appear in the environment.

Single lambda abstraction As it follows from lambda elimination, a lambda abstraction can be compiled as an equation having one argument. The compiled logic variable is bound to a newly generated unique name as

[2]The actual syntax of FPL does not contain multiple equations. We will still discuss the compilation of such definitions because equations play crucial role in our translation. In FPL selection is expressed by *if-then-else* while recursion is defined by using *letrec*. We compile these constructs via equations.

combinator. The same combinator will be used in the equation. The free variables of the lambda abstractions are collected as described in Section 2.1.2 and the closure of the free variables will be part of the combinator. As the next step, the equation is compiled. The goal and the program of the abstraction is simply the goal and the program from the compilation of the equation.

The abstraction `\x x+y` compiles to a bound logic variable `V7=unique_1(Y)` carrying the (introduced) name of the function, an empty goal `G7=true`, and the the program with one clause.

```
P7={apply(unique_1(Y),X,V7) -apply(add,X,V71),apply(V71,Y,V7) }
```

The free variable of the abstraction is provided in a closure.

Nested lambda abstractions. Nested lambda abstractions are also easy to turn into equations and then compiled as equations. For example the definition of the function *compose*

```
compose = \x y z. x (y z)
```

will be compiled as an equation

```
compose x y z = x (y z)
```

and has the result as

```
P8={ apply(compose_1,X,compose_2(X))
     apply(compose_2(X),Y,compose_3(X,Y))
     apply(compose_3(X,Y),Z,V8) - apply(Y,Z,V81),
       apply(X,V81,V8)
   }
```

with an empty goal and the logic variable V8 is bound to compose_1

Another example with subexpressions

```
(\x y 3 + (\z u z*v) x y)
```

This compiles to

```
P9={ apply(unique_2,X,unique_3(X))
     apply(unique_3(X),Y,V91)  - apply(add,3,V92),
       apply(unique_4,X,V93), apply(V93,Y,V94),
       apply(V92,V94,V91)
     apply(unique_4,Z,unique_5(Z))
     apply(unique_5(Z),U,V96)  - apply(mul,Z,V97),
       apply(V97,U,V96)
    }
```

with an empty goal, and the logic variable has the value as unique_2

5.2.5 Compiling ISWIM syntax

The compilation of ISWIM syntax can be derived from the compilation rule of lambda abstractions regarding its translation to lambda calculus (see Section 2.3.1). This solution, however, would lead to unnecessarily long code. We will consider certain shortcuts via equational definitions.

Dealing with if-then-else As an ISWIM construct *if-then-else* has a straightforward translation to lambda calculus (see Section 2.3.1). Compiling transformation of *if-then-else* using lambda abstraction would lead to very long code and less efficient evaluation however. Restricting the conditions to expressions where local definitions can not appear, the compilation

of ISWIM conditionals can be much simplified. If we only allow boolean valued operators ($<, >, ==$, etc) to appear in the condition part of *if-then-else* then the two choices in a conditional expression would be treated as choices of multiple equations. So we only have to compile the arguments of the conditional operators, which are simple lambda expressions and can be compiled by the scheme we have. For example `x>1` would compile to

```
apply(big,X,V10),apply(V10,1,V101), V101==true
```

where `big` is the curried combinator for `>`. The compiled code of the condition will be part of the clause compiled from the *then* part of the conditional expression. The whole *if-then-else* expression is compiled to multiple Prolog clauses.

The boolean operators 'and', 'or', and 'not' are compiled to a curried combinator form similar to the basic arithmetic operators.

Local let definitions. *Let ... in* definitions may contain constants and non-recursive functions. Local constant definitions can be stored in the environment if they are evaluated at the place where they occur. It can be done if local definitions can not appear in a lambda abstraction which could introduce unbound variables into the body of the local definition. The following expression

```
let mustnot = \x let a = x+x in 2*a in mustnot 5
```

is forbidden, because the inner *let ... in* expression is captured by the '\' operator.

The clauses resulting from the compilation of local function definitions are moved out to the global level. Applying the same restrictions for local

function definitions this change in the status of the definitions causes no trouble. However, their identifiers are only valid — or to be more precise only have the defined meaning — in the given scope. The combinators of local definitions need no closure. The identifier-value pair is added to the environment, but it is removed after the compilation (see Figure 5.1).

For example the following local definition

```
let succ x = x+1 in succ 1
```

compiles to

```
P10={apply(succ_1,X,Z) - apply(add,X,Z1), apply(Z1,1,Z) }
G10= apply(succ_1,1,V20)
```

and the result is returned in `V20`.

Local letrec definitions. Lambda elimination and lambda introduction were only defined for abstraction without recursion. Recursive definitions would need the Y combinator — or in our case a strict fixed-point combinator. Although the Y combinator itself is a lambda abstraction, and therefore can be translated to equation by means of lambda elimination [6], its compilation would result in extremely long code. This code would be needed every time, when a recursive expression occurs. This is not necessary, since we can obtain far better result using equations. To achieve this local recursive definitions can not occur inside lambda abstractions either.

Mutual recursion. Lambda elimination was not defined for mutually recursive functions which may contain free variables. If the expression does not contain further lambda abstraction, then the situation is similar to the expressions the lambda-lifting procedure by Johnsson [17] is designed for. That

algorithm requires $O(n^3)$ 'basic' operations and $O(n^2)$ 'set' operations where n is the number of function definitions. If the mutual *letrec ... in* expression is the outermost expression of an application, then they can be lifted out to the global level and be treated as any other equation. They cannot be returned as results, however.

5.3 Returning the value of an expression

After reading in and compiling an expression, the next step is to compute and return its value. At the end of Section 5.1 a general relation was introduced how to obtain the value of an expression from its compiled code. In this section some further details are discussed since now we know more about the compiled Prolog code.

When `G=true`, then the variable `V` must be bound as the result of the compilation. If `V` is bound to a constant or to a term, then that is returned as the result of the evaluation. There is no need to call the Prolog interpreter. If `V` is bound to a combinator, then there are two cases to consider. The combinator belongs either to a global definition or to a local definition.

For a global definition, the surface syntax name corresponding to the combinator is looked up in the environment and returned as the value of the evaluation. For example, simple typing in the name of a global definition would cause the name itself to be returned as a value.

The case of a local definition is more complicated. What can we say about the following application?

```
let tw=\f.\x.f (f x) in tw
```

The surface syntax name of a local definition would not mean anything on

the global level, so it can not be returned as value However, using lambda introduction, i e lambda-elimination in reverse order, we can obtain and return the lambda abstraction form of a local definition Or, for that matter, we could save (temporarily) the lambda abstraction format itself The use of Y combinator as lambda abstraction for returning recursive functions would make the definitions unreadable Obtaining the lambda form of locally defined mutually recursive functions would be especially difficult

When `G` is a query other than `true`, the Prolog interpreter is invoked to evaluate that query The value of the evaluation of the expression is then returned by the compiled logic variable `V`

After returning the value of an expression, the interpreter is ready for the next input

Chapter 6

Implementation and examples

We provide an implementation of FPL based on the compilation developed from the translation method

6.1 FPL: a functional language and programming environment

The original implementation of FPL includes an expression reader, a parser, a type-checker, an SECD code compiler, and an SECD emulator All programs are written in Prolog We use them courtesy of Dr Cheng

FPL has the programming tools we need to develop a system This assures a quicker and simpler prototype implementation, since we do not have to write and test the necessary system components Furthermore, the implementation of FPL is written in Prolog which meets our goal to have the whole integration implemented in Prolog We can replace the recent evaluation component with our approach of compiling and running Prolog code This allows us to concentrate on the implementation of the compilation and to carry out some experiments with it By implementing FPL based on our

approach, we can investigate the features of the compilation of functional programs to Prolog code in practice and test the effectiveness of the resulting code As a further gain FPL provides several additional features (macros, user defined operators etc) The advantage is that FPL helps us to see how the syntax of the functional part of LELP could be defined for better user convenience It also helped to realize necessary syntactical restrictions

6.1.1 Main features of FPL

There are two kinds of input to FPL, definitions and applications A definition must be preceded by the keyword *let*, a recursive definition must have a *rec* immediately following the *let* All other inputs which are not definitions are to be taken as function applications, i e expressions to be evaluated All functions in FPL are curried Every input expression has a type The user does not need to define the type of functions, instead, the system infers from the definition a most general type, which may be polymorphic

A few additional features create a simple yet effective programming environment The programming environment allows the user to load predefined functions from a file It is also possible to save the SECD code of the loaded definitions It is possible, but not convenient, to include the code of predefined functions when the system is started The implementation of these services can easily be modified to support the implementation of FPL based on our compilational approach

6.1.2 Programming examples

To demonstrate the new implementation of FPL a simple programming session is presented Between each user input and system answer given is the

'hidden' Prolog code resulting from the compilation preceded by '%'. This is the code which is either asserted in case of a definition or called as a Prolog query in case of an application. '[>' and '<]' are the system prompts.

```
cscs% fpl
TYPE fpl (version 2 - with compiled Prolog code)
Type ''help '' for help
[> let twice f x = f (f x)

% apply(twice_1,X,twice_2(X))
% apply(twice_2(X),Y,Z) - apply(X,Y,Z1),
%         apply(X,Z1,Z)

val twice = fn : (X -> X) -> (X -> X)

[> let succ n = n + 1

%apply(succ_1,X,Z) - apply(add,X,Z1),apply(Z1,1,Z)

val succ = fn : num -> num

[> twice twice twice succ 1

% ?- apply(twice_1,twice_1,Z1),apply(Z1,twice_1,Z2),
%         apply(Z2,succ_1,Z3),apply(Z3,1,Z)

val it = 17 : num

[> let rec fac n = if n == 0 then 1
<]                 else if n>0 then n * fac (n-1)
<] else error('negative input')
```

```
% apply(fac_1,X,1) :- apply(eq,X,Z1), apply(Z1,0,Z2),
%        Z2==true,!.
% apply(fac_1,X,Z) :- apply(eq,X,Z1), apply(Z1,0,Z2),
%        Z2==true,!, apply(mul,X,Z3), apply(sub,X,Z4),
%        apply(Z4,1,Z5), apply(fac_1,Z5,Z6), apply(Z3,Z6,Z).
% apply(fac,X,error('negative input')).

val fac = fn : num -> num

[> fac 10;

% ?- apply(fac,10,Z).

val it = 3628800 : num

[> fac (-10);

% ?- apply(fac,-10,Z).

negative input
val it = error : string

[> del "fac";

% Three clauses - defining the functions 'fac' - are removed.
```

6.2 Implementation issues

The SECD emulator of FPL is not needed any more. Its job is done partly by our lambda to clauses compiler and partly by the Prolog interpreter itself.

The SECD code compiler is replaced by a compiler for Prolog code. The expression reader does not require any changes. It is necessary to modify slightly the parser and the type checker in order to fit better to the new compiler. The implementation follows the strategy outlined in the previous chapter.

The implementation has to handle:

- the declaration as a Prolog program,
- the environment with the identifiers of built-in and predefined functions,
- the functional expression to be compiled and evaluated,
- the three components of the compiled Prolog code: the logic variable, the query, and the program.

Since the implementation is written in Prolog, a predicate is used to define the compilation scheme. Another predicate initiates the resulting query to compute the answer and this predicate returns the value as well. The declaration is loaded in when the system starts. Operators, macros and type definitions are handled by the programming environment.

The predicate `main` generates an infinite loop to read in and to evaluate user input until the user halts the system.

```
main  - read(Exp,T), init(Env), codeof(Exp,Env,V,G,P),
        add(P), evalof(V,G), print(V,T),',fail.
main  - main.
```

The communication with the user, reading expressions (`read(Exp,T)`) and printing the result and its type (`print(V,T)`) is provided by the pro-

gramming environment. The variable `T` carries information regarding the type of `Exp`. The environment is initialized by the `init(Env)` predicate. The predicate `add(P)` asserts the clauses in `P` to the beginning of the current declaration.

The relation `codeof(Exp,Env,V,G,P)` holds, if `V`, `G`, and `P` are the result of compiling the FPL expression `Exp` to Prolog code, given the environment `Env`. The expression is given as an abstract syntax tree represented by a list.

The relation `valueof(V,G)` holds, if after executing `G` as a Prolog query, `V` is bound to the value of the expression `G` and `V` are compiled from. The execution of the query requires the presence of the program which was compiled by the `codeof` predicate together with `V` and `G`.

The environment and the declaration The representation of declaration is straightforward and has been discussed already: it is provided as a Prolog program consisting of `apply/3` predicates.

An obvious choice to represent the environment is a list of identifier-value pairs. When a new variable is defined, the corresponding pair is added to the beginning of the list. This makes the search for identifiers easy and also supports proper scoping. For local definitions we will indeed use a list to represent bound variables paired with their values. For built-in operators and constants as well as predefined identifiers, however, we will use a binary predicate, `defined/2`, where the two arguments are the elements of the pair from the list. The advantage is twofold: we don't have to carry and pass around a large list, furthermore, looking up the value of an identifier needs less calls then searching trough the elements of a list one-by-one. (Since the global identifiers are unique, the optimization of WAM gives further efficiency

support)

6.3 Efficiency tests

[27] reported promising results with the term rewriting based function evaluation in Prolog In their tests they examined the rewrite-mimicking SLD-resolution and compared the results to Lisp systems doing the same task The above reference is our first source of information regarding the expected efficiency of evaluating functional expressions in Prolog based on the lambda elimination and the relationalization methods

In [27] and [7] a few benchmark tests were presented comparing the performance of certain Prolog and Lisp implementations The benchmarks included two tests on the list-processing performance of ALS-Prolog (Version 1 01), IBUKI Common Lisp (Release 01/01), and Franz Lisp (Opus 38 91)[1] The Lisp versions were run both in interpreted and compiled mode The Prolog test run as interpreted code The list-processing tests compared the performance of the included Prolog and Lisp systems for the basic computations

Furthermore, the systems were compared on the following application of the *twice* and *successor* functions

$$(((((twice\ \ twice)\ \ twice)\ \ twice)\ \ succ)\ \ 0)$$

For Prolog, the functions were tested in the form of equational clauses

```
twice F = g(F)
g(F) X = F (F X)
```

[1] ALS-Prolog is a trademark of Applied Logic Systems, IBUKI Common Lisp is a trademark of IBUKI, Franz Lisp is a trademark of Franz Inc

```
succ X = Y  - Y is X+1
```

The test was initiated by the following query

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

which is the Prolog equivalent of the above application

The tests were executed on a Sun3/280[2]. It was concluded that even considering the differences in efficiency between the Prolog interpreter and the Lisp systems for the basic computations, the Prolog version of the application runs just as fast as compiled Lisp.

Knowing the results from [27], we performed some benchmark tests comparing our compiled code and the ordinary Prolog version of the same functions expressed as relations.

6.3.1 Simple tests

Our first test[3] included the following definitions and application in FPL.

```
let twice  f  x = f  ( f  x )
let succ  x = x+1
twice twice twice twice succ 0
```

The compiled Prolog code is

```
apply(twice_1,F,twice_2(F))
apply(twice_2(F),X,V1) - apply(F,X,V2), apply(F,V2,V1)
```

[2]Sun is a trademark of Sun Microsystems Inc.

[3]All of our tests reported in this chapter were executed using ALS Prolog (Version 1.01) on a Sun3/280. Each test were run five times and the average CPU-time was calculated. The resolution of the timer was $1/60^{th}$ of a second.

```
apply(succ_1,X,V) :- apply(add,X,V1), apply(V1,1,V).
?- apply(twice,twice,V1), apply(V1,twice,V2),
   apply(V2,twice,V3), apply(V3,succ,V4), apply(V4,0,V).
```

There was only one built-in function included in the declaration

```
apply(add,X,add(X)).
apply(add(X),Y,V) :- V is X + Y.
```

For their Prolog code [27] reported 1.5 second as the best. Our compiled code needed 3.83 seconds to perform the same task.

Loading in the code of all the built-in functions — there are thirty five of them — gave a surprising result. ALS-Prolog run out of the default memory space. The problem is that some of the new definitions had multiple choices. The execution needed more space. It turned out that placing a cut to the beginning of the clauses of the *twice* function

```
apply(twice_1,F,twice_2(F)) :- !.
apply(twice_2(F),X,V1) :- !, apply(F,X,V2), apply(F,V2,V1).
```

reduces the required memory. Unfortunately, running the 'twice' benchmark with these clauses slowed the execution. The new execution time was 4.15 seconds when only the code for addition was given, and 6.07 seconds when all the built-in functions were included in the system.

6.3.2 Improving the efficiency of the compiled code

Comparing the code of the successor function provided by [27]

```
succ X = Y :- Y is X+1.
```

with our code for the same function

```
apply(succ_1,X,V) - apply(add,X,V1), apply(V1,1,V)
```

we can notice an important difference. Our compiled code uses a curried version of functions. However, when all arguments of a built-in function are provided, we can use the built-in Prolog version of the function. For example it is possible to compile to the following code:

```
apply(succ_1,X,V) - V is X+1
```

This code results in faster execution of the *twice* benchmark:

	one built-in function		35 built-in functions	
	curried	not curried	curried	not curried
without cut	3.83	1.72	aborted	aborted
using cut	4.15	1.90	6.07	2.82

Compared to the performance of the best functional languages, the results are not outstanding.

Although we can get better results allowing the compiler to omit currying whenever it is possible, the curried versions of the functions are still necessary. It allows the user to define functions like `let succ2 = + 2`.

6.3.3 Comparison to Prolog

To analyse the performance of our system we compared the compiled code to ordinary Prolog code as well. We prepared three benchmark tests. We wrote the FPL versions of the following functions: `fac` (*factorial*), `fib` (*Fibonacci*), and `ack` (*Ackermann*).

```
let rec fac x = if x==0 then 1 else x*fac (x-1)
let rec fib n = if n <= 2  then 1
                else fib (n-1) + fib (n-2)
let rec ack x y = if x==0 then y+1
                  else if y==0 then ack (x-1) 1
                  else ack (x-1) ack x (y-1))
```

We compiled these functions in two different ways. First, the functions were compiled using the curried versions of the built-in functions. Second, the built-in functions were replaced with their built-in Prolog counterpart as optimized code. Cut was not used and all built-in functions were provided. The same functions were also expressed in ordinary Prolog code as they are usually given as relations. We compared the execution time of the three different codes: the normal compilation, the optimized compilation, and the relational ones. The results shown in the following table are normalized to the relational definitions. The higher the number, the slower the execution was.

	compiled FPL		relational
	curried	optimized	Prolog
factorial	7	1.8	1
Fibonacci	7	1.5	1
Ackermann	7-8	2	1

These numbers suggest that, at least for the optimized version, the compilation is an acceptable technique and can be used to implement the proposed integration. However, further improvements are certainly necessary in order to provide an implementation of LELP where the user can choose between functions and relations and still get the same efficiency.

6.4 Summary

Due to the type-checking algorithm the recent implementation of FPL does not support multiple equations, mutually recursive functions on the global level, and constant pattern matching. We also have to note that certain restrictions apply on the form of local definitions and conditions. The compilation of these constructs was discussed in the previous chapter. Both the analysis of the compilation and our FPL implementation suggests that the equational form is more natural for our implementation method. Compiling from equations is more favorable since the translation is built on the equational form of function definitions. It is recommended to use multiple equations to define selection and recursion.

Although these restrictions apply, the functional expressions we can compile and thus include in the functional part of the integration are rich enough and provide powerful programming.

Chapter 7

Conclusions

There are classes of problems that can be expressed more naturally in a logic language than in a functional language and vice versa. Therefore, the integration of the two paradigms is not only of theoretical but of practical interest as well. Several solutions exist for the integration.

An integrated language would be especially valuable if it did not require any special evaluation mechanism, yet it could provide the efficiency of logic and functional language implementations.

We have proposed an integration method which introduces user-defined functions into Prolog. An implementation of the functional part based on a translation was suggested. We have summarized a method of translating lambda expression into a set of equations introduced in [6], called lambda elimination. We have also shown, how equations can be used to obtain the relational form of functions by means of the relationalization axiom as it was discussed in [27].

We examined the questions of putting these theoretical results into practice. A functional language was implemented to demonstrate the feasibility of our approach to implement the integration solution. We have described

how a compilation of this functional language to Prolog clauses can be used to evaluate functional programs. The compilation combines the methods of lambda-elimination and relationalization. The evaluation of the functional language using the compilation can be implemented in Prolog entirely. The implementation reported in this thesis was suitable to carry out some experiments. The resulting functional program evaluation is powerful and efficient, although there is room for improvement regarding execution time.

We demonstrated how an integrated language based on our solution could handle functional programming features like

- lambda abstraction,
- higher order functions,
- recursive functions,
- pattern matching

in a logic programming framework.

7.1 Future work

The development of the fully integrated language requires further work. Major changes are necessary for the expression reader and parser to meet the requirements of the mixed syntax. We have to talk about the problems that the presence of logic variables in lambda expressions would create. To extend the current implementation to handle the logic component would require certain changes. A tool is needed to read in the integrated language and to pass the functional components to the Prolog code compiler. It is also necessary to merge the original Prolog part and the compiled clauses together,

and to deposit the resulting code into a file ready to be read by any Prolog interpreter.

Functional programs are deterministic, while logic programs are not. In this sense the implementation of Prolog is more powerful than it is needed for the evaluation of a functional program compiled to Prolog code. There is no real need for the general unification when handling functional expressions, so the full power of logic programming and of WAM is not required. An interesting question concerns, how WAM could be modified to handle functions instead of, or as well as, relations. It would be possible to make certain evaluations even more efficient in both memory space and execution time.

The functional language evaluation supported by our compilation is strict. It is certainly worthwhile to investigate how WAM could be changed to support lazy evaluation or at least delayed evaluation of arguments.

Bibliography

[1] H Aït-Kaci *Warren's Abstract Machine* MIT Press, 1991

[2] H P Barendregt *The Lambda Calculus Its Syntax and Semantics* North-Holland, 1984

[3] M Bellia and G Levi The relation between logic and functional languages A survey *The Journal of Logic Programming*, 6(3) 217–236, 1986

[4] M Bruynooghe and M Wirsing, editors *Programming Language Implementation and Logic Programming '92* Springer-Verlag LNCS 631, 1992

[5] R M Burstall, D B MacQueen, and D T Sanella *Hope an experimental applicative language* Department of Computer Science, University of Edinburgh, 1980 CSR-62-80

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

[7] M H M Cheng, M H van Emden, and B E Richards On Warren's method for functional programming in logic In D H D Warren and

P Szeredi, editors, *Logic Programming Proceedings of the Seventh International Conference*, pages 546–560 MIT Press, 1990

[8] M H M Cheng and K Yukawa AP An assertional programming system *Advances in Logic Programming and Automatic Reasoning*, 1 120–134, 1992

[9] A Church A set of postulates for the foundation of logic *Annals of Math* , 2(33) 346–366, 1932

[10] K L Clark and S Gregory *PARLOG A parallel logic programming language* Imperial College of Science and Technlogy, London, 1983 Research Report DOC 83/5

[11] H B Curry et al *Combinatory logic* North-Holland, 1958

[12] D DeGroot and G Lindstrom, editors *Logic Programming Relations, Functions and Equations* Prentice Hall, 1986

[13] A J Field and P G Harrison *Functional Programming* Addison-Wesley, 1988

[14] M Hanus A functional and logic language with polymorphic types In *Proc Int Symp on Design and Implementation of Symbolic Computation Systems*, pages 215–224 Springer-Verlag LNCS 429, 1990

[15] J R Hindley and J P Seldin *Introduction to Combinators and λ-Calculus* Cambridge University Press, 1986

[16] R J M Hughes *The design and implementation of programming languages* PhD thesis, Programming Research Group, Oxford, 1984

[17] T Johnsson Lambda lifting transforming programs to recursive equations In Jouannaud J-P , editor, *Conference Functional Programming Languages and Computer Architecture, Nancy*, pages 190–203 Springer-Verlag, 1985 LNCS, vol 201

[18] H J Komorowski QLOG - the programming environment for Prolog in Lisp In *Logic Programming*, pages 315–322 Academic Press, 1982

[19] W A Kornfeld Equality for Prolog In *Proceedings of the Seventh IJCAI*, pages 513–519, 1983

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

[21] C Mellish and S Hardy Integrating PROLOG in the POPLOG environment In *Implementations of PROLOG*, pages 147–162 Ellis Horwood, 1984

[22] D A Miller, editor *Proceedings of the Workshop on the Lambda Prolog Programming Language, Philadelphia, Pennsylvania, July 1992* University of Pennsylvania, 1992

[23] D A Miller and G Nadathur Higher-order logic programming In *Proceedings of the Third International Conference on Logic Programming*, pages 448–462 The MIT Press, 1986

[24] J J Moreno-Navarro and M Rodriguez-Artalejo Logic programming with functions and predicates the language BABEL *Journal of Logic Programming*, 12(3) 191–223, 1992

[25] S L Peyton Jones *The Implementation of Functional Programming Languages* Prentice Hall, 1987

[26] U S Reddy Narrowing as the operational semantics of functional languages In *Proceedings of 1985 Symposium on Logic Programming*, pages 138–151 IEEE, Boston, MA, 1985

[27] B E Richards Contributions to functional programming in logic Master's thesis, University of Victoria, 1990

[28] J A Robinson and E E Sibert LOGLISP Motivations, design and implementation In *Logic Programming*, pages 299–314 Academic Press, 1982

[29] H Rogers Jr *Theory of Recursive Functions and Effective Computability* McGraw-Hill, 1967

[30] M Sato and T Sakurai Qute a functional language based on unification In D DeGroot and G Lindstrom, editors, *Logic Programming Relations, Functions and Equations*, pages 131–155 Prentice Hall, 1986

[31] G L Steele and G J Sussman *The Revised report on Scheme* MIT, 1978 AI Memo 452

[32] P A Subrahmanyam and J -H You FUNLOG = functions + logic a computational model integrating functional and logical programming In *Proc of IEEE International Symposium on Logic Programming*, pages 144–153, 1984

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

[34] D A Turner. Miranda — a non-strict functional language with polymorphic types. In Jouannaud J-P., editor, *Conference on Functional Programming Languages and Computer Architecture, Nancy*, pages 1–16. Springer-Verlag, 1985. LNCS, vol 201.

[35] 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.

[36] M H. van Emden and K. Yukawa. Logic programming with equations. *The Journal of Logic Programming*, 2(4):265–288, 1987.

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

VITA

Surname	Csaki	Given Names	Csaba
Place of Birth	Makó, Hungary	Date of Birth	11 Oct 1963

Educational Institutions Attended

University of Victoria	1991 to 1993
Technical University of Budapest, Hungary	1983 to 1988

Degrees Awarded

Master of Engineering in Mech Eng Tech Univ of Budapest 1988

Honours and Awards

'People's Republic Fellowship' by the Ministry of Education	1986 and 1987
'Scientific Student Circle' Special Award by the Rector of Tech Univ of Budapest	1987
'Excellent School Achievement and Trade Work' Award by the Dean of Fac of Mech Eng , TUB	1988
Scholarship of the Hungarian Academy of Sciences	1989-1991
University of Victoria Fellowship	1991-1993
Faculty of Graduate Studies Scolarship, UVic	1991

Publication

1 Cs Csáki, "Knowledge-based Program to Specify the Clamping of Rotational Parts" (in Hungarian), Thesis for the Master of Engineering, Tech Univ of Budapest, Hungary, Nov 1988

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: COMPILING A FUNCTIONAL LANGUAGE TO PROLOG

Author: [redacted]
(Signature)

CSABA CSÁKI
(Name in Block Letters)

08/09/93
(Date)