Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

CFC: AN EFFICIENT STREAM-PROCESSING ENVIRONMENT

A. Bostani

December 1992
CSD-920099

UNIVERSITY OF CALIFORNIA

Los Angeles

CFC: An Efficient Stream-Processing Environment

A thesis submitted in partial satisfaction of the

requirements for the degree Master of Science

in Computer Science

by

Arman Bostani

1992

The thesis of Arman Bostani is approved.

David F. Martin

David Martin

Richard R. Muntz

D. Stott Parker, Committee Chair

University of California, Los Angeles

1992

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor, D. Stott Parker for introducing me to logic programming and Prolog. His patience and guidance throughout the long process of completing this work was invaluable. Next, I would like to thank my committee members, Richard Muntz and David Martin for their comments during the writing of this work. I have also benefited greatly form the continual encouragement and support that I have received from my family.

In addition, I would like to thank all my friends at UCLA for the countless number of interesting discussions. Finally, my sincere thanks to Richard Stallman and the people at the GNU Project without whose wonderful tools, this work would not have been as enjoyable.

ABSTRACT OF THE THESIS

CFC: An Efficient Stream-Processing Environment

by

Arman Bostani

Master of Science in Computer Science

University of California, Los Angeles, 1992

Professor D. Stott Parker, Chair

This thesis describes the CFC, an efficient stream-processing environment. Stream-processing applications are developed using F*, a new programming language which combines logic programming, rewriting and lazy evaluation. Our primary focus in this work is to develop an environment for the efficient execution of F* programs and, where necessary, to provide extensions to the F* language itself.

In the course of this research, we have developed a compiler that translates a class of F* programs (called DF*) into instructions for an abstract machine, called DFAM. We show that it is possible to directly translate DFAM programs into C programs which are extremely portable and efficient.

CF* is introduced as a novel extension to the C programming language, providing a non-deterministic function call mechanism. We show that general F* programs can be compiled into an extension of DFAM, called FAM. A compiler

is described which compiles FAM programs into CF*. Furthermore, we show that it is possible to efficiently implement non-deterministic control structures, such as those found in CF*, on conventional machine architectures.

Finally, we have extended F* to make it suitable as a general purpose programming language. Also, unlike the early implementations of F*, this extended F* programming environment does not rely on the availability of Prolog as a host environment.

CHAPTER 1

Background

In recent years, much research has been directed at developing programming environments that efficiently combine various programming paradigms such as logic programming, rewriting, functional programming and lazy evaluation. It has been envisioned that such environments would provide the "expressive power of both functions, and relations" [Nar 88].

In [Nar 88], Narain proposes a new language, F*, and its implementation in Prolog, called Log(F), which can be used to do lazy functional programming in logic. In Log(F), an F* compiler translates F* rules into Horn clauses using an additional primitive for lazy simplification of F* terms, called *reduce.*

This chapter presents some background information about F* and Log(F) which have been used extensively at UCLA [Liv 88, Muntz 88, Parker 88a, Parker 88b] for the implementation of stream processing systems. We also review Bop, which is an extension to the Log(F) programming environment. Finally, we discuss related research in the area of compiling logic programming languages.

1.1 Overview of F* and Log(F)

This section provides an overview of the F* programming language and its implementation under Prolog, called Log(F) [Nar 88], which has been described as "a approach for combining logic programming, rewriting, and lazy evaluation".

F* programs are written as a set of rewrite rules of the following form:

```
LHS => RHS.
```

where LHS and RHS are terms satisfying a group of restrictions described later on in this section.

The following example illustrates how one would write an F* program similar to the Prolog "append/3" predicate:

```
append([], L) => L.
append([X|L1], L2) => [X|append(L1,L2)].
```

Using the Log(F) compiler described in [Nar 88], this program is translated into something similar to the following Prolog code:

```
reduce(append(L1, L2), L3) :-
        reduce(L1, []),
        reduce(L2, L3).
reduce(append(L1, L2), L3) :-
        reduce(L1, [X|L]),
        reduce([X|append(L,L2)], L3).

reduce([], []).
reduce([X|L], [X|L]).
```

Having translated the F* "append" rules into Prolog, we can append two lists (i.e. `[1,2,3]` and `[4,5,6]`) using the "reduce" predicate:

```
?- reduce( append([1,2,3], [4,5,6]), X ).
```

With the above query, we receive the following instantiation for the variable X:

```
X = [1 | append([2,3], [4,5,6])].
```

The reader may have noticed that the F* append works "lazily". That is, only the head of the resulting list is computed. The tail of the list can be further reduced when necessary. This method of demand driven computation is termed *lazy evaluation.*

Note that in the translated F* rules, reducing a list (e.g. `[]`, `[1, 2]`) will simply return the list itself. Terms with this property are called *constructor terms* and their functors are called *constructor symbols.* We will use expressions of the form c(a1,...,an) to denote constructors and F* functions (e.g. "append") will be denoted by f(a1,...,an).

The following is the set of restrictions placed on F* rules:

a) LHS is not a variable. This is similar to the restriction in Prolog that the head of a goal cannot be a variable.

b) LHS is not of the form c(t1,..,tn) where c is a constructor. This restriction provides a simple halting condition for the rewriting process.

c) Given an LHS of the form, f(t1,t2,...,tn), then each ti is either a variable or a term of the form c(X1,...,Xm) where c is an m-ary constructor symbol, and Xi are variables. This restriction was introduced in order to simplify the theoretical analysis of F*.

This restriction has been relaxed in our implementation of F*. For an LHS of the form, f(t1,t2,...,tn), each ti is a *constructor term.* A *constructor term* is defined to be either a variable or a term of the form c(t1,...,tm) where c is an m-ary constructor symbol, and ti are constructor terms.

d) There is at most one occurrence of any variable in LHS. This restriction assures that when reducing a term f(t1,...,tn) against the LHS of a rule f(L1,...,Ln), we need only match each ti with Li. That is, function arguments can be unified independently (potentially in parallel [Liv 88]). We believe that it is possible to relax this restriction as well. Rules with the following form which violate restriction (d):

```
f(A,A) => RHS.
```

can always be rewritten as:

```
f(A,B) => if(eq(A,B), RHS, fail).
```

e) All variables of RHS appear in LHS. This ensures that reductions never produce non-ground terms.

Like Prolog, F* is a non-deterministic language. Where, in Prolog, a given goal may evaluate to *true* more than once, an F* function may have more than one return value. For instance, given the following F* code segment:

```
f => 10.
f => 20.
```

the function `f` has two return values, 10 and 20. In Log(F), the rules for `f` are translated to the following reduce rules:

```
reduce(f, 10).
reduce(f, 20).
```

Therefore, the reduction of `f` using the Prolog goal, `reduce(f, X)`, will succeed twice, instantiating variable X to the value 10 and then to 20.

1.2 Related Research

In the past few years, much attentions has been focused in the area of compiling logic programs. The majority of current Prolog compilers are in some way related to the work presented by Warren in [Warr 83]. Warren describes a method for the compilation of Prolog, through the use of an intermediate code for a virtual machine referred to as the Warren Abstract Machine (WAM).

In a similar approach Mellish [Mell 85] describes the compilation of Prolog into a procedural language with first-class continuations, called POPLOG. Mellish also argues that the majority of Prolog programs are directed and deterministic and can therefore be directly compiled into efficient code running on conventional machines.

In [Bruy 86], Bruynooghe suggests compiling Prolog to Pascal in order to improve garbage-collection performance in Prolog. Although the main motivation of this paper is developing static garbage collection mechanisms for Prolog, his work provides interesting insights into how Prolog-like languages can be translated into

procedural languages.

Weiner and Ramakrishnan [Wein 88] describe the implementation of "piggy-back" Prolog compiler which translates Prolog into C. Much like our work, it is an attempt to prove that Prolog (or logic programming languages in general) can be efficiently mapped onto a conventional architecture. Their Prolog compiler relies heavily on user supplied annotations to generate efficient code. Non-determinism is implemented using a continuation-passing mechanism similar to [Mell 85].

Boyd and Karam [Boyd 90] describe a "dual" Prolog and C programming environment. Prolog programs are converted to C using a method similar to [Wein 88]. Non-determinism is handled through the usage of a continuation list. Emphasis is placed on the potential for the hand-tuning of C code generated by the translator.

There have also been several attempts at embedding logic programming in functional languages [Hayn 86, Kahn 84, Sriv 85, Stic 88, Tsan 88]. In particular there have been many attempts at embedding logic programming in Lisp like languages. Kahn and Carlsson [Kahn 84] describe two implementations of Prolog on Lisp machines. Backtracking is supported through the use of continuations.

Haynes [Hayn 86] gives a taxonomy of embeddings and introduces the notion of "complete" embeddings. He goes on to show how logic programming control states (e.g. during depth first search) can be implemented using Scheme's first-class continuations. Haynes also argues that language support for first-class continuations is essential for embedding logic programming in that language.

Srivastava [Sriv 85] also describes the embedding of logic programming in an

extended Scheme language, called Scheme/L. A similar approach to [Hayn 86] is used in the implementation of backtracking. The primary emphasis of this work, however, is on the efficient implementation of logic variables in Scheme/L.

Researchers have also developed new programming paradigms which combine various aspects of functional and logic programming [Ders 85, Levi 87, Tama 84, van E 87]. Tamaki [Tama 84] describes a language which combines rewriting and logic programming. This is accomplished through the introduction of a reducibility predicate similar to the Log(F) *reduce*. In effect, the reducibility predicate introduces a form of extended unification to logic programming.

Dershowitz and Plaisted [Ders 85] extend logic programming through the usage of conditional rewrite rules. In [Levi 87], K-LEAF is described as an experimental language based on extending Horn Clause Logic with equality. van Emden and Yukawa [van E 87] also propose a methodology for extending logic programming mechanisms to include functional programming through the introduction of equality. They argue for the implementation of equation solving as a special case of SLD resolution.

1.3 Bop

In [Parker 92], Parker describes the Bop programming environment. Like our work, the development of Bop was motivated by the need for an efficient and flexible stream processing environment. In contrast with the CFC, however, Bop was designed as a portable extension to Prolog.

The Bop environment combines programming paradigms such as logic programming, conditional rewriting, narrowing, functional programming and lazy evaluation. Bop's syntax and semantics are similar to that of F*, but two major differences exist.

Firstly, Bop provides the ability to write conditional rewrite rules. These rules have the following form:

```
LHS => RHS :- Condition.
```

where LHS and RHS are terms which satisfy similar rules as those of F*. *Condition* is a Prolog term that must be satisfied before the RHS is reduced. In essence, this approach provides the capability of writing conditional rewrite rules.

Another important feature of Bop is its ability to perform *narrowing* which is a form of term rewriting with logic variables. In Bop, restriction (e) in F* has been removed. Therefore, it is possible for Bop terms to contain variables and for variables to become bound when rules are applied. For instance, consider the rule:

```
power(X, 0) => 1 :- dif(X,0).
```

With Bop, there is a *narrowing*

```
power(2,P) =>> 1
```

that binds the variable P to 0. With F*, however, there is no reduction for the term `power(2, P)` since it is not a ground term.

As with Log(F), Bop rules are translated into Horn clauses. The Bop compiler, however, makes much greater use of clause indexing features of modern Prolog

systems to produce more efficient Prolog code. We should also note that many of the ideas presented in this work on the compilation of F* can be applied to compiling Bop.

CHAPTER 2

Introduction

2.1 Purpose of this thesis

Stream processing systems, such as those being developed at UCLA, need to be able to handle large quantities of data distributed over many processors on a network. The data often takes the form of time series, and stream operations can range anywhere from applying simple stream operators to performing heavy numerical computations.

In the implementation of F* in the Log(F) environment, F* rules are translated to Prolog clauses. This implementation provides a flexible environment where Prolog and F* programs can be easily combined. Unfortunately, in many cases, F* programs developed under the Log(F) environment, do not provide the performance necessary for the efficient implementation of stream processing.

This thesis explores the feasibility of compiling F* programs into a more conventional programming language such as C. By compiling F* programs into C, our goal is produce small, portable, high performance modules that can be used as the building blocks for an efficient stream processing system.

Also, in our experience with programming in the Log(F) environment, we have

noticed that F* has primarily been used as an "embedded" rather than the primary language. That is, for any given program, most of the code is usually written in Prolog and only small portions of the code which require lazy evaluation or rewriting have been developed in F*. This is in part due to the performance problems mentioned earlier and the lack of important features in the language which we will describe throughout this thesis.

In developing the CFC, our purpose has been to extend F* in order to make it suitable as a general purpose programming language. Our objective is to create a total, integrated environment which does not rely on the availability of Prolog as a host environment.

2.2 Why Compile to C?

As previously mentioned, we did not feel that the Log(F) implementation of F* provided sufficient performance for the implementation of an efficient stream processing environment. This is primarily due to the fact that all F* rules are translated into `reduce` clauses rather than separate Prolog predicates. Also, to run even a small F* program, we would need to execute a Prolog interpreter which is a rather large executable[1].

Two other alternatives exist for the implementation of F*:

1. Translation of F* to C or native machine code.

[1]Version 0.6 of the SICStus Prolog interpreter uses approximately 750kb of main memory upon startup.

2. Interpretation of F* programs using an F* interpreter.

Several questions should be explored in order to decide which alternative is best:

1. Is there a need to interface to other languages on the machine? (i.e. providing a simple interface for other languages to call routines in our language.)

2. What is the usual size of the programs that we will write and execute in the language? If these programs are relatively small, the excessive size of an interpreter can be a significant burden.

3. How well do the data structures in the language match those of the host machine?

4. How well do the control structures in the language map to the architecture of the machine?

In the following sections we examine these questions with respect to the implementation of F*. The reader is forewarned, however, that like most difficult problems, there are no clear-cut answers.

2.2.1 External Interfaces

None of the Prolog interpreters that we know of implements calling Prolog predicates from external languages efficiently[2]. This is one the major drawbacks

[2]Several implementations of external interfaces to Prolog interpreters exist. These implementation, however, rely on starting a Prolog subprocess with which they communicate via some

to using interpreters. It is, in general, very difficult to provide an interface whereby interpreted functions (predicates) may be called from an external language.

With compiled code, however, providing a callable function interface is as easy as providing a layer for the mapping of caller data types into those of the callee, and vice versa.

2.2.2 Program Size

Since to run interpreted programs, we always need to execute the interpreter, we must always pay the space penalties for loading the entire interpreter. Therefore, if the interpreter is bulky, we will need to spend considerable time on startup and memory usage even if the program to be interpreted is comparatively small.

Since we envisioned stream operators to be small F* functions, possibly operating as a separate processes, the overhead of the interpreter would be significant.

It must be noted, however, that for executing large programs, the space advantage is with interpreters. As we will discuss in the next section, compiled code is generally much larger than the byte code representation used by many interpreters.

2.2.3 Data Structures

Conventional programming languages such as C usually have data types which are very similar to those supported by the underlying hardware (e.g. pointers, integers, floats, etc.). The similarity of the language data types to those supported

form of interprocess communication (e.g. TCP/IP, RPC, etc.). Thus, these interfaces tend to be much less efficient than the usual calling interface supported by the host architecture.

by the host architecture enables compilers to produce efficient code for programs written in that language.

With a language such as F*, however, which supports lazy evaluation and complex terms, the mapping of language data types to the host architecture is not as straightforward. A significant amount of code must be generated to create and manipulate the language data types. In an interpretive environment, the code for handling language data types is localized within the interpreter itself. In a compilation setting, however, the code for data manipulation is duplicated each time these data types are accessed.

Compilation, therefore, leads to a significant amount of bulk in the final compiled executable. In a comparison of compiled versus interpreted Prolog code, [Kral 87] reports that his compiled Prolog code used ten times the amount of memory used by his interpreted byte-code representation. This problem is a major disadvantage of compilation.

2.2.4 Control Structures

As with data types, if the language control structures match those of the host architecture, more efficient code can be produced by a compiler for that language.

Current conventional machine architectures are geared towards executing procedural languages. One of the major problems in compiling languages such as F* has been the mapping of non-deterministic control constructs to underlying procedure oriented architectures. We will show that it is possible to efficiently map

non-deterministic control constructs to conventional machine architectures. We believe this to be one of the major contributions of this thesis.

2.2.5 Portability and Performance

One of the key advantages of interpretive systems over traditional compilers which generate native code is portability. Interpreters are generally written in high level languages and in many cases contain a small core of hand written assembly code, and are therefore relatively easy to port. With traditional compiler technology, however, a great amount of work is necessary to re-target the compiler for a new machine architecture.

By compiling to C rather than machine code, we can achieve instant portability without the sacrifice in performance usually associated with interpreters. Also, due to the popularity of the C programming language, instruction sets for the current generation of microprocessors have been especially optimized for the execution of C programs. Of course, the quality of the generated machine code depends substantially on the C compiler used.

CHAPTER 3

Compiling Deterministic F*

3.1 Determinacy in Prolog

Since the inception of Prolog, researchers have tried to create more efficient Prolog implementations by introducing user-supplied annotations [Mell 85, Newt 87, Turk 86, Wein 88]. These annotations (i.e. mode, type, domain declarations) provide a Prolog compiler with hints about the user's intended usage of Prolog predicates. Using these annotations, a Prolog compiler is able to generate more efficient code leading to significant improvements in performance [Wein 88].

Determinacy detection is an important aspect of such optimizations. Using the information provided by the user, a Prolog compiler can determine whether a given predicate is able to succeed more than once (i.e. whether it is non-deterministic).

The usage of such annotations, however, tend to be rather cumbersome. Annotated Prolog code is often harder to maintain. If any code is modified, its annotations must be accordingly modified. Also, it may be necessary to write duplicate code to achieve these optimizations. For instance, an annotated Prolog "append" predicate can be written as:

```
:- mode append(+, +, -).

append([], L, L).
```

```
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

The mode declaration `append(+, +, -)`, specifies that the append predicate is to receive non-variable terms for the 1st and 2nd arguments and that the 3rd argument will be a variable. If we needed an append with a different mode specification (e.g. `append(-, -, +)`), we would either have to do away with the mode declaration or duplicate the append predicate and rename it.

3.2 Determinacy in F*

In [Nar 88] Narain proves that there exists a class of F* programs, called **deterministic F* (DF*)** which possess certain useful computational properties, such as confluence and directedness. F* programs with these properties are guaranteed to return at most one value (i.e. they are deterministic).

A DF* program is defined as an F* program P satisfying two new restrictions:

f) Let LHS1 and LHS2 be variants of heads of two rules in P, such that LHS1 and LHS2 have no variables in common. Then LHS1 and LHS2 do not unify.

g) Let $f(L1, ..., Li, ..., Lm) \Rightarrow RHS$ be a rule in P, where Li is not a variable. Then in all other rules $f(K1, ..., Ki, ..., Km) \Rightarrow RHS$ in P, Ki must be non-variable.

In the following examples the rules for `member` do not satisfy restriction (f) and are therefore not a DF* program.

```
member([X|L]) => X.
member([X|L]) => member(L).
```

The rules for **reverse** and **rev**, however, satisfy both (f) and (g) and therefore constitute a valid F* program.

```
reverse(L) => rev(L,[]).

rev([], L) => L.
rev([X|L1], L2) => rev(L1, [X|L2]).
```

In contrast to the user-specified annotations introduced for optimizing Prolog, detection and specification of determinism in F* is simply a matter of checking the (f) and (g) syntactic restrictions. Later in this work, we will show that the detection of determinacy in F* programs can considerably improve their performance.

3.3 DFAM: The DF* Abstract Machine

In [Warr 83] David Warren presented a new method for compiling Prolog through the use of an intermediate code for a virtual machine referred to as the Warren Abstract Machine (WAM). Since its introduction, the WAM has evolved and has become the basis of many Prolog implementations [Gabr 85, Newt 87, Turk 86].

In the following sections, we define an abstract machine, called the DF* Abstract Machine (or DFAM), which we use as intermediate representation for compiling DF*. Although the DFAM is not directly derived from the WAM, many similarities exist.

Like the WAM, the DFAM provides facilities for the definition and calling of procedures. Procedure arguments are passed on a global stack, and the data structures of the DFAM are allocated from a global heap.

The major differences between the WAM and the DFAM arise in the handling of variables. Many of the complexities associated with keeping track of variable instantiations are eliminated since there is no variable binding in F*. There are also considerable differences in the handling of non-determinism in our abstract machine. These differences will be further explained in the following chapter.

3.4 Design Principles

In this section we describe the basic design principles that led to the current definition of the DFAM.

3.4.1 Compilation of DFAM to C

An important driving force in the design of the DFAM was providing the ability for the translation of the DFAM to C. Therefore, the DFAM has a strong resemblance to conventional procedural languages. For instance, groups of DF* rules which have the same principal functor and arity are compiled into a single DFAM function which in-turn is translated into one C routine.

Alternatively, it would have been possible to compile each individual DF* rule into a single DFAM function. For instance, given the following DF* program :

```
father('John') => 'George'.
```

```
father('David' => 'John'.
father('Jill') => 'David'.
```

With this method, we would generate a DFAM function for each of the above rules. Therefore, to reduce a DF* term such as `father(X)`, where the value of X is determined at runtime, we would need to call three DFAM functions.

This approach would dramatically increase the number of DFAM function calls. Since each DFAM function is translated into a C function, the number of C function calls would increase accordingly. Unfortunately, in conventional architectures, function calls are known to be rather inefficient when compared to other instructions. Our initial prototypes using this mechanism were on average 3 times slower than our current solution.

3.4.2 Simple Instruction Set

In the past two decades, research into computer architecture design has indicated that it is more effective to develop machines with smaller and simpler instruction sets (RISC) rather than computers with more complex set of instructions (CISC). This phenomenon is, in part, due to the following reasons:

- It is easier to optimize and debug RISC instruction sets simply because there are fewer instructions. Also, RISC instructions tend to be simpler than those found in CISC architectures, and therefore easier to implement.

- It is usually harder to develop compilers that effectively use all instruction in a CISC computer. Even when such a compiler is developed, it tends to be

harder to debug due to the added compiler complexity.

Although RISC principles have primarily been applied to design of hardware architectures, we believe that the aforementioned rationale also applies to the design of the DFAM. The DFAM, therefore, has been developed with a very small set of rather simple instructions.

3.4.3 The DFAM's Data Types

The DFAM's data types and the way its data structures are manipulated are very similar to those used in SICStus Prolog. Indeed, several DFAM instructions which manipulate data structures, directly resemble WAM instructions. Compatibility with the SICStus Prolog data types allows us to easily integrate the DFAM with the Prolog environment. This approach provided us access to many facilities available in SICStus Prolog which are, as of yet, not implemented in the CFC. The motivation for the usage of SICStus Prolog data types in DFAM will become clearer when we discuss the architecture of the CFC in Section 5.1.

3.5 The Basics

There are two basic types of objects in the DFAM:

Data: These objects are used for the storage of atomic data types which include integers, floats, and atoms. The following are instances of atomic data objects: 1776, 7.04, 'Bell'. Data objects can also store list and structure descriptors.

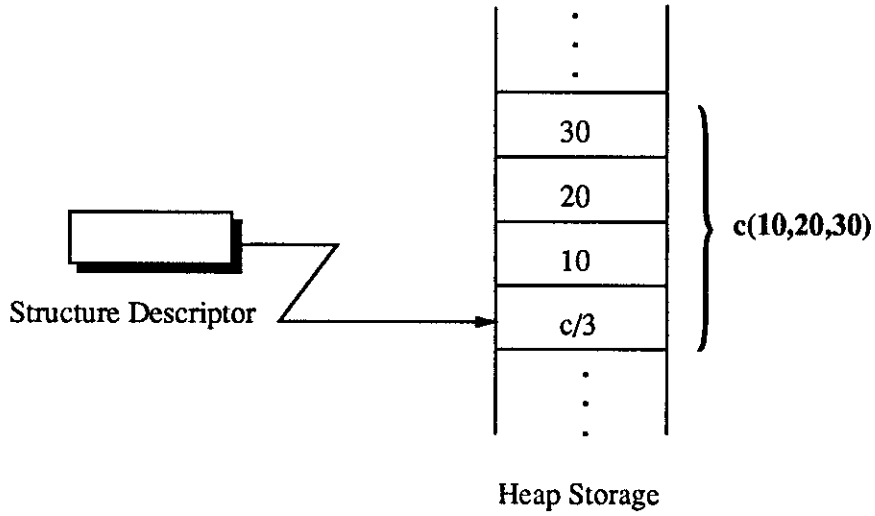

Figure 3.1: Allocation of `c(10,20,30)` on the DFAM heap.

These descriptors are used to refer to more complex structures stored on the DFAM global heap.

Address: These objects are used to store the addresses of *data* types. They are similar to pointers in a conventional programming language such as C. *Address* types are used in the construction and traversal of DFAM data structure stored on the heap.

3.6 Structures and Lists

DFAM constructs its data structures by storing objects of type *data* on the global heap. Two structured types are supported:

Structures These are non-atomic objects such as the F* term: `c(10,20,30)`. Figure 3.1 displays the representation of this structure in the global heap.

`c/3` represents a combination of the functor and arity of the term. The term's arguments appear, in order, in consecutive heap cells following the functor/arity pair.

A structure descriptor (which is of type *data*) can be used to refer to this structure on the heap. In this implementation, structure descriptors are simply pointers to the heap location where the functor/arity pair is stored.

Lists As in Prolog, F* lists are structures with "." as the head functor and an arity of 2. List structures appear quite frequently in many F* application. Indeed, in both Bop and Tangram, Lists have been used as the primary data structure for the implementation of streams. As in SICStus Prolog, list structures have been implemented differently from other structures for to attain better performance.

In F*, a list is made up of two basic data items, the head and the tail. For instance, consider the term `[10 | 20]`, which can also be written as `'.'(10, 20)`. Figure 3.2 displays the representation of this term in the heap.

Since with lists, the functor/arity pair is always `'.'/2`, this pair is not stored on the heap. Instead, list descriptors simply refer to the head element of list. The list's tail is assumed to be stored in the next heap cell.

To illustrate how longer lists can be created on the heap, consider the list `[1, 2]` which can also be written as `'.'(1, '.'(2, []))`. Figure 3.3 displays the representation of this list in the heap.

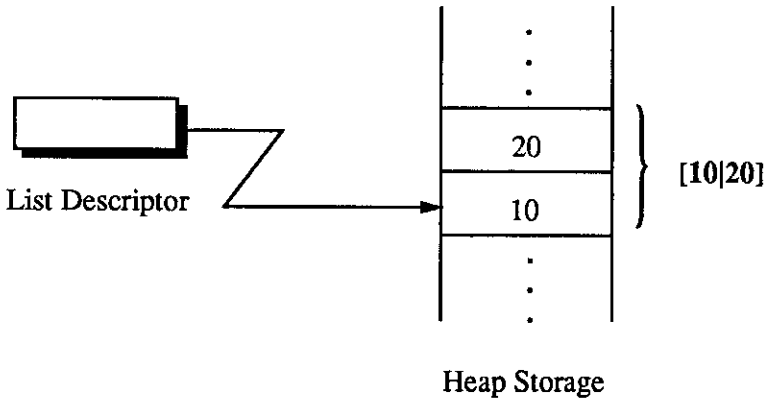

Figure 3.2: Allocation of `[10 | 20]` on the DFAM heap.

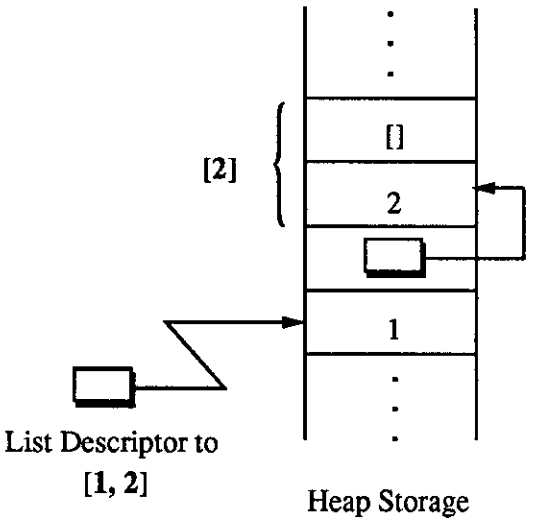

Figure 3.3: Allocation of `[1, 2]` on the DFAM heap.

3.7 Local Storage

Each DFAM function can have three types of local storage (local registers):

Function Arguments These local registers are of type *data*, and are used to store the arguments passed to the current function. The register x_N, is used to access the Nth argument to the current function. In DFAM, all function arguments are "input". That is, function arguments are not modified by the callee.

Local Data Registers Data registers may contain any object of type *data*. The register d_N, is used to access the Nth local data register.

These registers are usually used for storing intermediate results of computations. For instance, a data register can be used to save the return value from a call to a DFAM function or result of a reduction of a DFAM term.

Local Address Registers These registers are of type *address* and contain pointers to *data* locations. The register a_N, is used to access the Nth local address register. These registers are similar to pointers in a conventional programming language and are usually used when traversing DFAM data structures stored on the heap.

The DFAM local registers are analogous to local variables in C. They exit only for the duration of the function, and they are accessible only within the current function.

The DFAM does not limit the number of registers used in a function. The number of local registers provided depends on the resources available on the host system, and is therefore dependent on the given implementation.

3.8 Heap Related Instructions

In this section we introduce the DFAM instructions which are used to create and manipulate F* structures on the heap. Note that unlike the WAM, some of the DFAM instructions are functional and can return values. Return values form these instructions can then be placed in registers or sent as arguments to other instructions.

d_N = D Assigns data value D to data register d_N.

a_N = A Assigns address value A to address register a_N.

push(D) Pushes D, which is an object of type *data* onto the the heap, incrementing the heap-top counter. This is the only DFAM instruction that actually allocates storage on the heap. For instance, consider the following F* rule:

```
f => c(10,20,30).
```

Given that in the above rule, "c" is a constructor symbol, every time the term "f" is reduced, we return the structure `c(10,20,30)`. This structure can be constructed in heap at runtime using the following DFAM code segment:

```
push(c/3)
```

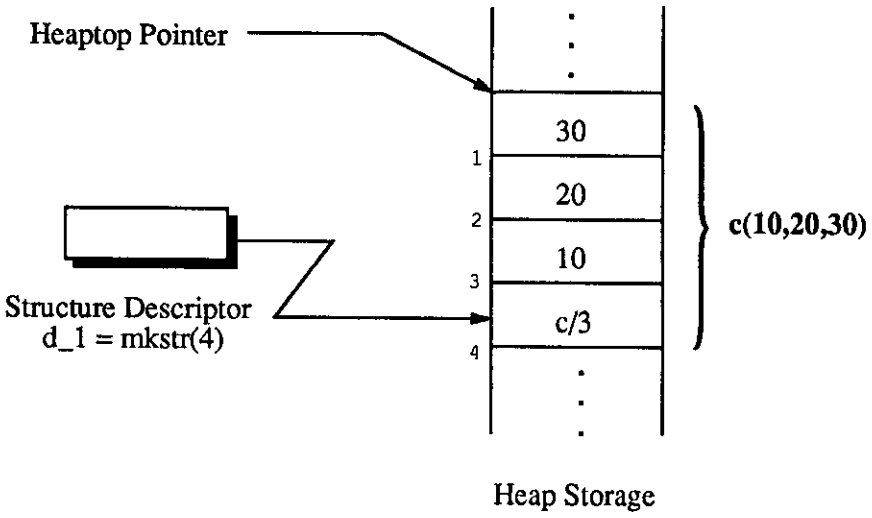

Figure 3.4: `d_1` refers to the allocation of `c(10,20,30)` on the DFAM heap.

```
push(10)
push(20)
push(30)
```

The resulting structure on the heap is displayed in Figure 3.1.

mkstr(N) Returns a structure descriptor which identifies a structure whose head functor is N cells from the top of the heap. Therefore, after executing the **push** instructions in the previous example, `mkstr(4)` would return a structure descriptor for the term `c(10,20,30)` on the heap. For instance, consider the following assignment:

```
d_1 = mkstr(4)
```

After this assignment, the data register `d_1` contains a reference to the structure created on the heap. Figure 3.4 displays the result of this assignment.

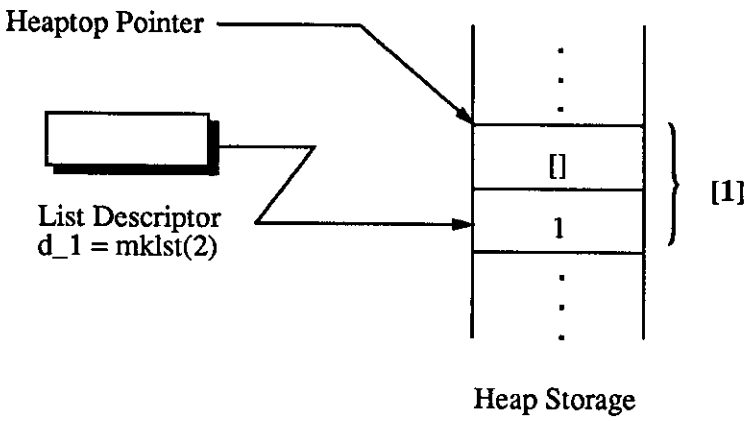

Figure 3.5: `d_1` refers to the allocation of `[1]` on the DFAM heap.

mklst(N) Returns a list descriptor for a list whose head element is N cells from the top of the heap. This instruction is very similar to **mkstr** and is used to create descriptors that refer to lists stored on the heap. For instance, consider the following code fragment:

```
push(1)
push([])
d_1 = mklst(2)
```

After the execution of this code fragment, `d_1` contains a reference to the newly created `[1]` list on the heap. Figure 3.5 displays the result of this assignment.

tostr(R) Given a structure descriptor D, returns the address of the structure's head functor. This instruction is used to translate a structure descriptor into a physical address in the heap where the structure is stored. This address can then be used to access the various components of the structure. For instance, given the following code:

```
push(c/1)
push(100)
d_1 = mkstr(2)
a_1 = tostr(d_1)
```

After the execution of this code fragment, `d_1` will contain a structure descriptor which refers to the term `c(100)` on the heap. Also, the address register `a_1` will contain the address of the functor/arity pair, `c/1`, on the heap.

tolst(D) Given a list descriptor D, returns the address of head element of the list. This instruction is very similar to `tostr` and is used to translate a list descriptor into a physical address in the heap where the list is stored.

***A** Returns the *data* value stored at *address* A. This is similar to a pointer dereference in a conventional programming language such as C.

next(a_N) Advances the address stored in local address register a_N to point to the next heap address. This instruction is used in traversing structure elements. Consider the following code fragment:

```
push(1)
push([])
d_1 = mklst(2)          % d_1 refers to list [1]
a_1 = tolst(d_1)        % a_1 contains address to list
d_2 = *a_1              % d_2 gets head of list, 1
next(a_1)               % increment a_1
d_3 = *a_1              % d_3 gets tail of list, []
```

After the execution of this code fragment, the list `[1]` is created on the heap with the list descriptor `d_1`. The `a_1` register contains the address of the head

of the list on the heap after the `tolst` instruction is executed. Subsequently, `d_2` is assigned the contents at address `a_1`. Since `a_1` points to the head of the list, after the execution of `d_2 = *a_1`, `d_2` is assigned the value 1. After the `next` instruction, `a_1` in incremented and points to the next cell in the heap, the tail of the list. Therefore, the final assignment in the code fragment, sets the value of `d_3` to `'[]'`.

goto(L) Transfers control to code at label L. This instruction is similar to a *goto* statement in a conventional programming language.

test(C, Op) If condition C is not true, perform operation *Op*. Currently, the compiler generates only `goto` instructions for *Op*. Refer to the next section for an explanation of conditional expressions in DFAM.

3.9 Conditional Expressions

The following is the list of conditional expressions that can be used in conjunctions with the `test` instruction:

D1 == D2 Evaluates to true if and only if $D1$ and $D2$, both of type *data*, are equal.

isstr(D) Evaluates to true if and only if D is a structure descriptor.

islst(D) Evaluates to true if and only if D is a list descriptor.

These instructions are primarily used in the implementation of F* unification and are further explained in Section 3.11.

3.10 DFAM: Call/Return Instructions

As was previously mentioned, the DFAM instruction set is very similar to a procedural programming language. Groups of DF* rules which have the same principal functor and arity, are compiled into a single DFAM function.

To illustrate the compilation of DF* into DFAM instructions, consider the following simple DF* rule:

```
square(X) => times(X,X).
```

The `square` rule reduces to the multiplication of its argument, X, with itself. `square` is compiled into the following DFAM function:

```
square(x_1)
[
        data    d_1;            % declare local variable d_1
        d_1 = times(x_1,x_1);   % d_1 = x_1*x_1
        dreturn(d_1);           % return d_1 as result
]
```

The DFAM function `square`, takes one argument represented by `x_1`. The `data` statement declares a single local data register `d_1`. Subsequently, the `times` function is called, the result of which will be the multiplication of `x_1` by `x_1`. The result is placed in the `d_1` register and returned using the `dreturn` instruction.

The following is the set of instructions used in the DFAM calling mechanism:

F(D1, ..., Dn) Call DFAM function F sending it arguments $D1$, ..., Dn. Every such function call will return a value of type *data*.

dreduce(D, d_N) Reduce data value D and put the resulting value in data register d_N. Note that if D is a function term, the function is called and the result is placed in d_N. Otherwise, we simply copy D into d_N.

dreturn(D) Return data value D from the current DFAM function. Control is passed to the caller.

3.11 Unification

To implement term matching in F*, we generate a sequence of test-and-branch DFAM instructions. For instance, given the following F* rules:

```
father('Tom') => 'Dick'.
father('Dick') => 'Harry'.
```

we generate the following code for the DFAM "father" function:

```
father(x_1)
[
        data    d_1;
        dreduce(x_1,d_1)
        test(d_1 == 'Tom', goto(father0))
        dreturn('Dick')
father0:
        test(d_1 == 'Dick', goto(father1))
        dreturn('Harry')
father1:
        dreturn(fail)
]
```

In this function, the `test` instructions are used to check whether reduction of the argument `x_1` (placed in `d_1`) matches either the `Tom` or `Dick` atoms. The DFAM function above can be thought of as executing the following pseudo-code:

```
father(x_1)
{
        d_1 = reduce(x_1);

        if (d_1 == 'Tom')
                dreturn('Dick');
        else if (d == 'Dick')
                dreturn('Harry');
        else dreturn(fail);
}
```

When more complex term unification is required, the compiler will generate code that traverses the structure of the two terms being unified. For instance, given the following F* rule:

```
car([X|Y]) => X.
```

The following DFAM code is generated (note that anything appearing after a "%" is a comment).

```
car(x_1)
[
        addr    a_1;
        data    d_4,d_3,d_2,d_1; % local data declarations

        dreduce(x_1,d_1)         % reduce 1st arg into d_1

        % if d_1 not a list goto car0
        test(islst(d_1), goto(car0))
        a_1 = tolst(d_1)         % a_1 points to the list's head
        d_2 = *a_1               % d_2 <- head of list
        next(a_1)                % a_1 points to the list's tail
        d_3 = *a_1               % d_3 <- tail of list
```

```
        dreduce(d_2,d_4)        % reduce head of list into d_4
        dreturn(d_4)            % return d_4
car0:
        dreturn(fail)           % fail
]
```

As is evident from the comments in the DFAM code, the structure of the argument to the car function is traversed. If the argument is not a list, the function fails. Otherwise, d_2 will hold the list head and d_3 its tail. Obviously, the code that finds the tail of the list is not necessary. The d_3 register is never used after its assignment. This is one of the many possible optimizations not yet implemented.[1]

3.12 Function Terms

Lazy evaluation in F* leads to the occurrence of "lazy function call" terms (function terms for short). For instance, given the following F* program:

```
f(X,Y) => X.
g => [f(10,20)].
```

When g is reduced, the result of the reduction is not [10], but rather [f(10,20)], where f(10,20) is a function term invocation. Ideally, such a function invocation should be represented on the heap as a structure containing a reference to the function to be invoked and the list of arguments to be passed to the function upon invocation.

[1] Many optimizations that have been successfully applied too compiling Prolog programs, such as *indexing* and *tail recursion optimization* (see [Gabr 85, Warr 83]) can also be used in an F* implementation. The impact of such optimizations is discussed later in this work.

Unfortunately, since SICStus Prolog data structures are used for the implementation of DFAM terms, we were not able to directly implement function terms as distinct structures on the heap. This is, of course, due to the fact that SICStus Prolog does not support function terms.

In our current implementation, we represent function terms with terms whose head functor is `'LF'`. For instance, in the previous example, the term `[f(10,20)]` is represented as:

```
[ 'LF'(IndexOf_f, 10, 20) ]
```

"IndexOf_f" is an internal index that will be generated by the system and will be used to locate the DFAM function `f`. The last 2 structure arguments to `'LF'` are the arguments to be passed `f`, once the function term is called (i.e. reduced).

3.13 Translating DFAM to C

The translation of DFAM into C requires two components:

- Implementation of the DFAM data types in C.
- Implementation of a set of C macros to which individual DFAM statements can be translated.

3.13.1 DFAM Data Types in C

The *data* type in DFAM is mapped to type `DATA` in C which is simply a synonym for the 32-bit *long int* type. DFAM, however, is polymorphic. That is, an object

of type *data* may contain integers, floats, atoms, structure descriptors, etc. To implement DFAM's polymorphism in C, we use a 3-bit tag on `DATA` types. This tag allows us to determine the type of object stored in a `DATA` cell (i.e. whether it is an integer, float, atom, etc.).

DFAM objects of type *address* are represented by `ADDR` types in C. The `ADDR` type is defined in C as a pointer to an object of type `DATA`.

3.13.2 DFAM Statements in C

The translation of DFAM into C is simply a matter of providing a set of C macro definitions. All DFAM statements map directly into these C macros.

DFAM functions are directly translated into C functions, returning values of type `DATA`. Similarly, function call and return statements also map directly into C's call/return mechanism. For instance, given the following F* function:

```
father('Tom') => 'Dick'.
```

we generate the following DFAM function:

```
father(x_1)
[
        data    d_1;
        dreduce(x_1,d_1)
        test(d_1 == 'Tom', goto(father0))
        dreturn('Dick')
father0:
        dreturn(fail)
]
```

This DFAM function has the following C translation:

```
DATA father_1(x_1)
        DATA    x_1;
{
        DATA    d_1;
        DREDUCE(x_1,d_1);
        TEST(d_1 == STUB0, GOTO(father0));      % STUB0 is 'Tom'
        return(STUB1);                          % STUB1 is 'Dick'
father0:
        return(LOGF_FAIL);
}
```

The `father_1` function takes one argument, `x_1`. `x_1` is reduced into the local data register `d_1` using the `DREDUCE` macro. This macro is used to check whether or not its first argument is a DFAM function term. If so, the function is called and the result is placed in the second argument. Otherwise, the first argument is simply copied into the the second.

In the current implementation of the CFC, F* atoms are hashed. Therefore, when comparing atoms, we need only compare their hash values. In the C code for the `father_1` function, `STUB0` and `STUB1` represent macro definitions for the hash values of the "Tom" and "Dick" atoms respectively. The values of these macros are determined at load time by the CFC link editor, the function of which is explained in Section 5.2.1.

The `TEST` macro is used to match the function's argument with the "Tom" atom (or `STUB0`). The operation of this statement is equivalent to the following C code:

```
if (d_1 != STUB0) goto father0;
```

If unification succeeds, the atom "Dick" (represented by `STUB1`) is returned. Oth-

erwise, the LOGF_FAIL atom is returned (i.e. the father_1 function fails).

Compilation of the following F* rule illustrates heap allocation in DFAM and C.

```
cons(X, Y) => [X|Y].
```

This F* rule is translated to the DFAM function:

```
cons(x_1,x_2)
[
        push(x_1)
        push(x_2)
        dreturn(mklst(2))
]
```

This DFAM function can be translated to the following C function:

```
DATA cons_2(x_1,x_2)
        DATA    x_1,x_2;
{
        PUSH(x_1);
        PUSH(x_2);
        return(MKLST(2));
}
```

The PUSH macro operates on a "top of the heap" pointer. This pointer, called global_top, indicates the location of heap storage not presently allocated. Thus, PUSH(some_data), is equivalent to the C statement:

```
*global_top++ = some_data;
```

Therefore, cons_2 creates the list [X|Y] by allocating heap storage for X and Y, and returning a list descriptor to X on the heap. In the MKLST(N) macro, N is subtracted from the global_top pointer and the result is tagged as a list descriptor.

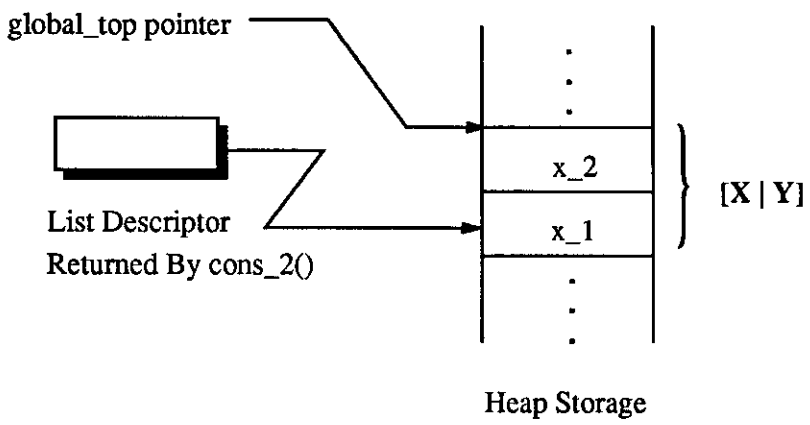

Figure 3.6: Allocation of `[X | Y]` on the CF* heap.

3.13.3 Function Terms in C

The CFC environment keeps track of all functions loaded into the system via the usage of a function table. When a new function is added, its address and name are stored in the function table and the function is given an index.

Previously we mentioned that function terms are represented using the following structure on the heap:

```
'LF'(IndexOf_Function, A1, ... An)
```

Where `A1, ..., An` are the function arguments, and `IndexOf_Function` is the index given to the function at load time. When a function term is reduced, the index is used to look for the function's address in the function table. Once the address is identified, the function is invoked with the arguments, `A1, ..., An`.

CHAPTER 4

Compiling Non-Deterministic F*

This chapter describes the compilation of non-deterministic F* programs in the CFC environment. The main topic of this chapter is defining extensions to the DFAM instruction set and the C programming language needed to support non-determinism.

4.1 FAM: The F* Abstract Machine

In this section we describe the compilation of non-deterministic F* into FAM, a superset of the DFAM instruction set. FAM data types and the instructions which manipulate data structures are the same as DFAM. FAM, however, contains extra control structures which permit the implementation of a non-deterministic function call mechanism.

4.1.1 Design Principles

To implement non-determinism in FAM, it is necessary to extend the DFAM function call mechanism such that functions are allowed to return more than one value. Therefore, when a non-deterministic function f is called, it needs to return a *continuation* along with the usual function return value. The *continuation* can

4.1.3 Non-Deterministic Terms

As was previously mentioned, F* terms may contain function call terms. Since such calls can also return multiple values, F* terms can also be non-deterministic. For instance, given the previous definition of the F* function g, the term `[g]` can be unified with either `[1]` or `[2]`.

Such non-deterministic terms can appear as arguments in function calls or as nested terms within larger terms. To illustrate how such terms are handled within FAM, consider the following F* rule:

```
h(10, 20) => 30.
```

In any given call to function `h` the first and the second arguments can be non-deterministic terms. Therefore, when reducing the arguments to `h`, we need to consider the possibility that such a reduction may return more than one value. The following FAM function is generated when `h` is compiled:

```
h(x_1,x_2)
[
        data    d_2,d_1;
        reduce(x_1,d_1)
           reduce(x_2,d_2)
              % check to see if reduced 1st arg = 10
              test(d_1 == 10, goto(h0))

              % check to see if reduced 2nd arg = 20
              test(d_2 == 20, goto(h0))

              % continue with return value 30
              cont(30)
h0:
           end_reduce
        end_reduce
```

```
        return(fail)
]
```

In the FAM function above, both `x_1` and `x_2` can be non-deterministic terms. Therefore, the compiler generates code which executes the RHS of `h` with all possible combinations of reductions for the first and second arguments. This is done by nesting the reduction block for the second argument inside the reduction block of the first.

4.2 Translating FAM to CF*

In this section we describe CF*, an extension to the C language, used in the implementation of FAM. CF* extends C by allowing functions to return more than one value. In CF*, functions can save their current state of execution before returning. This feature allows a CF* function to be called and subsequently restarted to continue where it "left off".

We believe that the addition of non-deterministic control to C is a is a major contribution of this work. We will show that such constructs can be easily and efficiently implemented on conventional computer architectures.

4.3 Non-deterministic Control Constructs

Non-determinism in CF* is provided through the use of the following new control constructs:

FORALL_CALL(F(A1,...,An), d_N) ... FORALL_END [1] This construct is analogous to the FAM `call/call_end` instructions. We iterate over all values returned from the CF* function F until there are no more return values available. The values returned are placed in the C variable, `d_N`, and the code block up to the **FORALL_END** instruction is executed. We repeat the call to F until there are no further values to be returned.

CONT(D) This construct returns the value D from the current function. The state of execution of the current function is preserved and, therefore, its execution may be resumed by the caller. If resumed, the current function continues executing as if it had never encountered the `CONT` instruction.

RETURN(D) Return value D from the current CF* function. This instruction is equivalent to the FAM `return` instruction.

FORALL_REDUCE(D, d_N) ... FORALL_END This construct is similar to the FAM `reduce` and `end_reduce` instructions. Since the term D can be a function term, the reduction of D may result in more than one value. Therefore, we iterate over every reduction of D, placing the result in `d_N`, and executing the code inside the `FORALL_REDUCE` block. The block is exited when there are no further reductions.

[1] The reader should note that, in practice, we have actually used a different syntax for the implementation of the `FORALL_CALL` statement. It suffices to mention, however, that the syntactic representation given here is simply a sugar-coated version of the actual syntax. The reason for not using this syntax in the implementation is due to the limitations of the C language preprocessor.

Due to the similarity of their control constructs, the translation of FAM to CF* is straightforward. For instance, consider the following FAM functions from a previous example:

```
g()
[
        cont(1);
        cont(2);
        return(fail);
]

f()
[
        data    d_1;

        call(g(), d_1);
           cont(d_1);
        end_call;

        return(fail);
]
```

The following CF* translation is produced:

```
DATA g_0()
{
        CONT(STUB0);              % STUB0 and STUB1 are macros for
        CONT(STUB1);              % atoms '1' and '2' respectively.
        RETURN(LOGF_FAIL); % Their function is explained later.
}

DATA f_0()
{
        DATA    d_1;

        FORALL_CALL(g_0, d_1);
           CONT(d_1);
        FORALL_END;

        RETURN(LOGF_FAIL);
}
```

4.4 Implementation of CF* Control Constructs

The CF* constructs (i.e. FORALL_CALL, CONT, RETURN, etc.) are implemented using a combination of C and assembly macro definitions. These macros allow us to enhance the C calling mechanism such that the state of any CF* routine can be saved during a `CONT` instruction, and then restored from the calling function.

4.4.1 CF* Procedure Call Mechanism

At any given point in the execution of a C function, the state of the function can be described using the following elements:

- The arguments with which the function was called.
- Local storage area used for storing local variables.
- Return address of the caller. This is the address to which this function returns.
- Values of the machine registers. These include all regular registers and the program counter (PC), stack pointer (SP), etc.

If, somehow, the function's state information can be saved at some point during its execution, it can be restarted to resume execution from the point that its state was saved.

In fact, most of the state information needed to save the state of a function is already available on the system stack. To understand our implementation of the CF* control constructs, we will briefly review the procedure calling mechanisms on conventional machine architectures.

4.4.2 Procedure Calling Conventions

The implementation of the C language procedure call mechanism is very similar among conventional machine architectures. For our purposes, it will suffice to explain the standard procedure calling mechanism on a Motorola 68000 processor.

Most architectures such as the M68k reserve a register which acts as a frame pointer (FP). This register is used to access local variables and procedure arguments on the stack, and eventually to retrieve the caller's return address. Therefore, the FP can be used to access much of the information needed to restore the state of a function.

Consider the following C code fragment:

```
f1()
{
        ...
        f2(a1, ..., an);
        ...
}

f2(x1, ..., xn)
{
        ...
}
```

When function `f2` is called from `f1`, the following operations are performed (Fig-

ure 4.1 illustrates f2's C stack frame generated by these operations):

1. `a1,...,an`, which are the arguments to function `f2`, are pushed onto the stack.

2. The current program counter (PC) and frame pointer (FP) are pushed onto the stack.

3. FP is set to point to the top of the stack (i.e. FP $\leftarrow$ SP).

4. Storage is allocated on top of the stack for `f2`'s local variables.

5. PC is set to the top of the `f2` function, and the execution of `f2` begins.

6. To return, `f2` places the return value in the return register (d0).

7. `f2`'s local storage is removed from the stack by resetting SP.

8. FP and PC are restored from the stack, and therefore control is passed to `f1` and `f2`'s arguments are popped from the stack.

9. The remaining code in `f1` is executed.

By step (5), `f2`'s frame has already been created on the stack, and is illustrated by Figure 4.1. The system FP is able to address `f2's` local storage, arguments and the saved FP and PC from the point at which `f2` was called. The saved PC and FP will be used to restore the caller's state when we return from `f2`.

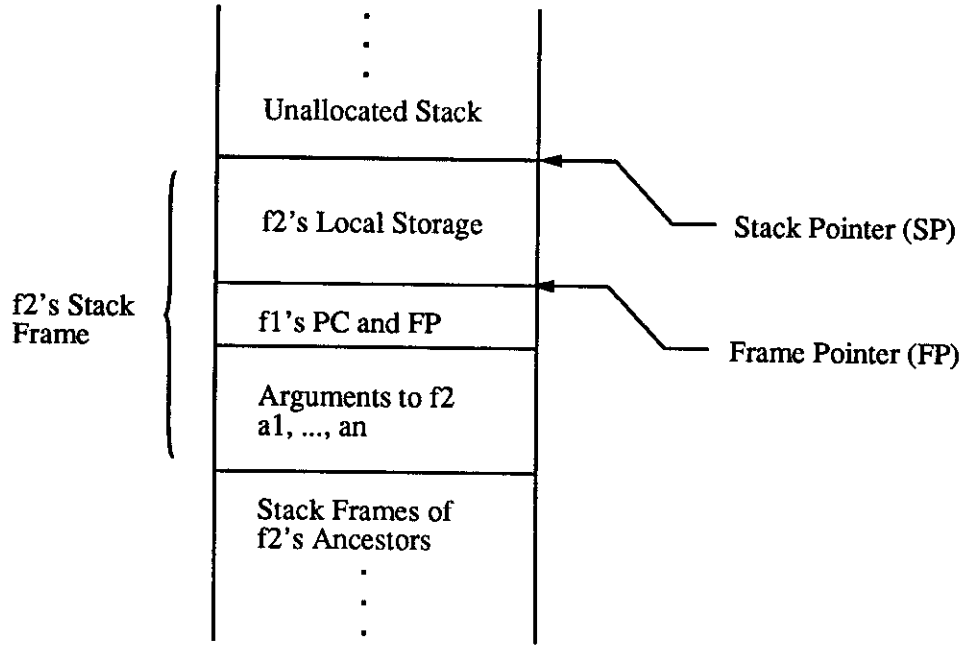

Figure 4.1: f2's stack frame on the C stack after being called from f1.

4.4.3 Non-Deterministic CF* Functions

In our previous example, the reader may have noticed that most of a function's state information, which was alluded to earlier, is available in the function's stack frame. The only missing state information is the current values of machine registers (i.e. PC, SP, and FP).[2]

In conventional procedural languages, procedure stack frames are created when a procedure is called, and are subsequently removed (popped) when the procedure returns. Therefore, control is always within the function whose frame is on top of the stack.

[2]Note that other general machine registers also constitute state information. For the sake of simplicity, we will delay discussion of these registers for later sections.

In CF*, however, we allow control to pass back to the calling function without removing the current function's frame from the stack. Thus allowing CF* functions to be restarted where they "left off". To see how the CONT instruction works, let's examine the following CF* code fragment:

```
f1()
{
        ...
        FORALL_CALL(f2(a1, ..., an), d_1);
                ...
        FORALL_END;
        ...
}

f2(x1, ..., xn)
{
        ...
        CONT(v);
f2_restart:
        ...
}
```

After `f2` is called from `f1`, the `CONT(v)` instruction in `f2` is executed and the following operations are performed:

1. The program counter (marked by the label `f2_restart`) is saved in the current frame.

2. The current FP and the value `v` are placed in return registers as return values from `f2`. The current FP is returned as a *continuation*, so that `f2` may be restarted at some later point.

3. Control is passed back to the calling function `f1`, so that it may resume its execution.

Figure 4.2 illustrates the state of the CF* call stack after the execution of the `CONT(v)` instruction.

4.4.4 The CF* FORALL_CALL Construct

As was previously mentioned, the CF* `FORALL_CALL` construct calls a given CF* function, and iterates over all its return values. For instance, consider the following CF* function:

```
DATA f1()
{
        DATA    d_1;

        FORALL_CALL(f2, d_1);
                ...
                ...     /* Some CF* Code */
                ...
        FORALL_END;

        RETURN(LOGF_FAIL);
}
```

The `FORALL_CALL` and `FORALL_END` macros are implemented using C and assembly macros. The following pseudo-code translation of the CF* code, above, illustrates the implementation of these constructs:

```
DATA f1()
{
        DATA d_1;

        /*
        ** d_1 is set to the value returned from f2. f2_fp is
        ** set to the frame pointer value which was also returned
        ** by the call to function f2.
        */
        <d_1, f2_fp> = f2();
```

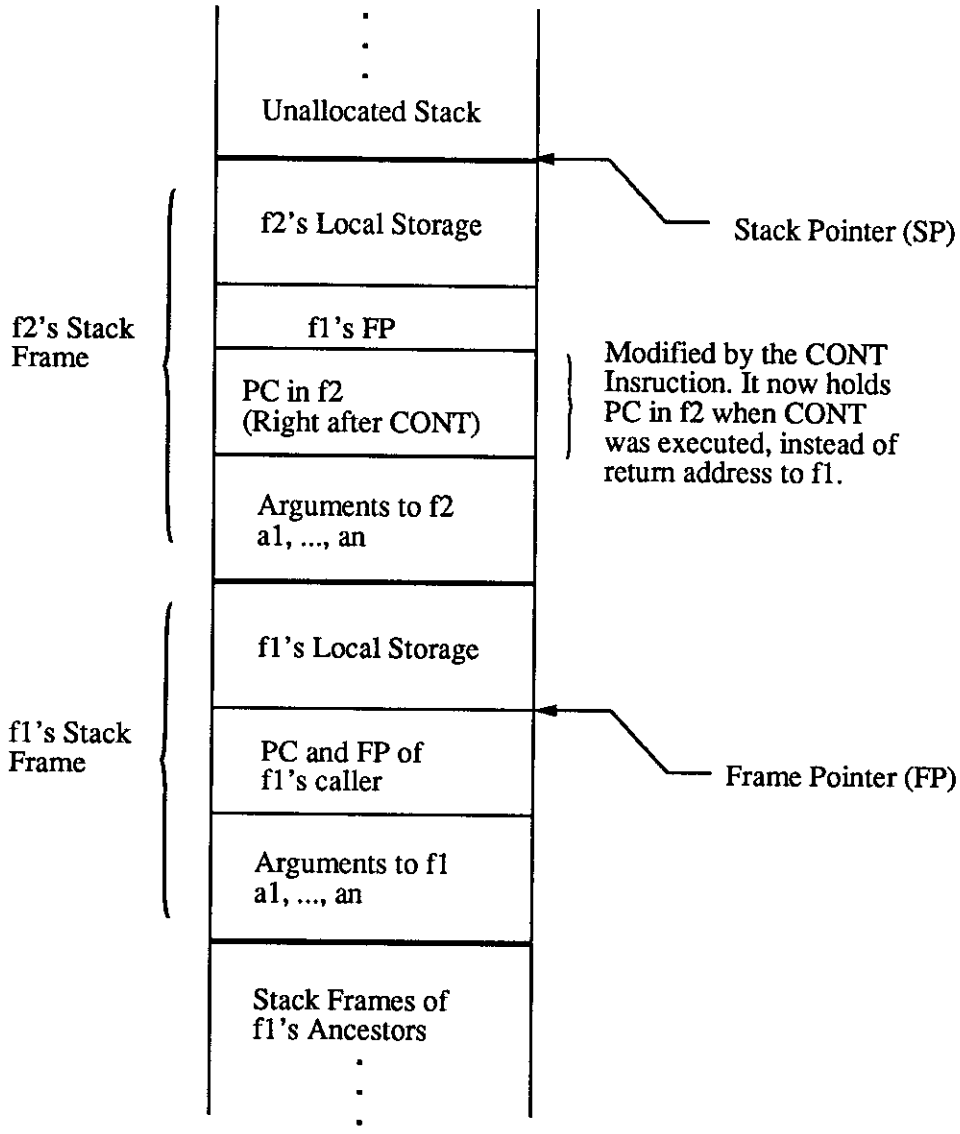

Figure 4.2: The state of the call stack after the execution of `CONT(v)`.

```
        while (d_1 != LOGF_FAIL) {
                ...
                ...     /* Some CF* Code */
                ...

                if (f2_fp == NULL_THREAD)
                        break;

                <d_1, f2_fp> = RECALL(f2_fp);
        }

        RETURN(LOGF_FAIL);
}
```

In the above example, when `f2()` is called and subsequently passes control back to `f1`, it returns two values. The return value of `f2` is placed in `d_1`. `f2` also returns its FP value which is placed in `f2_fp`.

If the return value from the called function is `LOGF_FAIL`, there are no further return values and we exit the call block. Otherwise the code within the call block is executed. Afterwards, `RECALL(f2_fp)` restarts the execution of `f2`. As with a call, a `RECALL` returns a `<value, continuation>` pair. When there are no further values to be returned from the call to `f2`, the call block is exited.

Before the `f2_fp` frame pointer is used to restart `f2`'s execution, we check to see whether `f2_fp` is has the value `NULL_THREAD`. If so, there are no further values from the function and we exit the call block.

4.4.5 The CF* RETURN Construct

As was previously mentioned, the CF* RETURN is used as an optimization. Like CONT, it too returns a <value, continuation> pair. The continuation, however, is always set to NULL_THREAD, indicating that the thread of execution for the called CF* function has terminated.

4.4.6 The CF* FORALL_REDUCE Construct

The CF* FORALL_REDUCE construct operates very similarly to FORALL_CALL. In a FORALL_REDUCE block, however, it is necessary to determine whether the object being reduced is a function term. If so, the function is invoked and we iterate over the values returned from the call. For instance, consider the following CF* function:

```
DATA f1(x_1)
{
        DATA    d_1;

        FORALL_REDUCE(x_1, d_1);
                ...
                ...     /* Some CF* Code */
                ...
        FORALL_END;

        RETURN(LOGF_FAIL);
}
```

The following pseudo-code translation of CF* code, above, illustrates the implementation of the FORALL_REDUCE blocks:

```
DATA f1(x_1)
```

```
{
        DATA d_1;

        /*
        ** Check to see if argument x_1 is a function term. If so
        ** we call it, assigning the value/continuation pair.
        ** Otherwise, x_1 has no further reductions and d_1 is set
        ** to x_1, and f2_fp is set to NULL_THREAD.
        */
        if (IS_LAZY_FUNCTION(x_1)) {
                <d_1, f2_fp> = LAZY_CALL(x_1);
        }
        else {
                d_1   = x_1;
                f2_fp = NULL_THREAD;
        }

        while (d_1 != LOGF_FAIL) {
                ...
                ...     /* Some CF* Code */
                ...

                if (f2_fp == NULL_THREAD)
                        break;

                <d_1, f2_fp> = RECALL(f2_fp);
        }

        RETURN(LOGF_FAIL);
}
```

The IS_LAZY_FUNCTION macro is used to check whether argument x_1 is a function term. If so, the function is invoked using the LAZY_CALL macro which behaves exactly like a direct call to a CF* function and so a <value, continuation> pair is returned. If x_1 is not a function term, it has only one reduction, and that is x_1. Therefore, we assign NULL_THREAD to f2_fp, indicating that there are no further reductions.

4.4.7 CF* Portability

We feel that the implementation of the CF* constructs on architectures other than the Motorola 68000 would also be straightforward. Our current implementation has been developed primarily in C and only 40 lines of M68k assembly were required to implement the non-deterministic function calls.

CHAPTER 5

Architecture of the CFC

The CFC system consists of two primary components, the runtime and compile time environments. F* programs are compiled to CF* using the F* compiler. CF* programs are then linked and executed within the CFC runtime environment.

5.1 The F* Compiler

The CFC compiler is responsible for the compilation of F* programs to FAM or CF*. The F* compiler, which has been written in Prolog, is divided into the following modules:

Syntax Verification The compiler reads in the F* rules and verifies adherence to restrictions (a) through (e) (see section 1)

Determinacy Detection The compiler determines whether restrictions (f) and (g) are met. If the program is deterministic, a flag is turned on inside the compiler which tells the code generator to produce deterministic FAM (DFAM).

LHS Processing In processing the LHS, the compiler determines the steps required for the unification of F* terms with the LHS. This involves possible

reduction of arguments, and performing the necessary pattern matching operations.

RHS Processing In compiling the RHS components of F* rules, the compiler determines the steps needed in calling functions and creation of data structures (on the global heap) to be returned from the current function.

Code Generation The code generation (or the back-end) component of the F* compiler is responsible for the translation of the compiler's internal representation of code to FAM or CF*. Based on a flag generated by the determinacy detection analysis, the back-end will generate either deterministic or non-deterministic code.

As of yet, the F* compiler lacks an optimization component. Such an optimizer could perform better register allocation, dead code elimination, tail recursion optimization, etc. We shall show later that these optimizations can greatly improve the performance of compiled F* programs.

5.2 The Runtime Environment

Early in the design phase of CFC, it was decided that we would build a prototype of the CFC environment on top of an existing Prolog system (SICStus Prolog). Such an implementations has several advantages:

Integration With Prolog: By building the our system on SICStus Prolog, we were able to provide an interface for the execution of F* functions from

Prolog. Therefore, much of the testing of CFC was done using the Prolog interface. Also, a simple F* shell was implemented using the Prolog interface.

Compatible Data Types Since compatibility was retained with SICStus Prolog data types in our F* implementation, many internal SICStus Prolog functions such as arithmetic, structure manipulation (e.g. `functor/3`, `''=..''`, etc.), database functions (e.g. `assert`, `retract`), etc., were automatically made available to F* programmers.

Dynamic Loading: SICStus Prolog provides support for dynamic loading of compiled object modules. This allowed us to dynamically load our compiled F* modules with the aid of a CF* link-editor written in Prolog.

Garbage Collection: Since the SICStus Prolog heap storage was used in storing F* terms, we were able to utilize the SICStus Prolog garbage collection mechanisms.

Using SICStus Prolog in the implementation of F* greatly reduced the time needed in developing support software, and freed us to experiment with F*. Such a "piggy-back" implementation is not without its problems, however. As previously mentioned, due to the fact that Prolog data representation was used to store F* terms, we were not able to directly represent F* function terms.

5.2.1 Runtime Link Editor

The CFC environment provides facilities for dynamically linking CF* functions to the runtime environment. The dynamic linker allows for loading of multiple CF* files by resolving external function calls and references to atoms. To perform this task, the CF* linker produces a "stub definitions file" which contains two sets of information:

Atom Definitions Since Prolog atoms are represented using their hash values, their values are not known until runtime. The F* compiler places *atom stubs* in the CF* file when generating references to atoms. The dynamic object linker determines the hash value for each atom at link time and generates a macro definition for each of these *atom stubs.*

Function Definitions The target function term is identified using its index into the CFC function table (see Section 3.13.3). Since the value of this index is not available until runtime, the compiler places *function stubs* in the places of a function index when generating function term. The linker determines the correct function index for each function loaded. It then generates a macro definition for each of these *function stubs.*

Having generated the stub definitions, the CF* runtime linker uses a C compiler to compile CF* files together with stub definitions into an object module.[1] Object

[1]Compilation of CF* files is performed by GNU C. The primary reason for the usage of this C compiler is the availability of a flexible mechanism for *inlining* assembly instructions in C code.

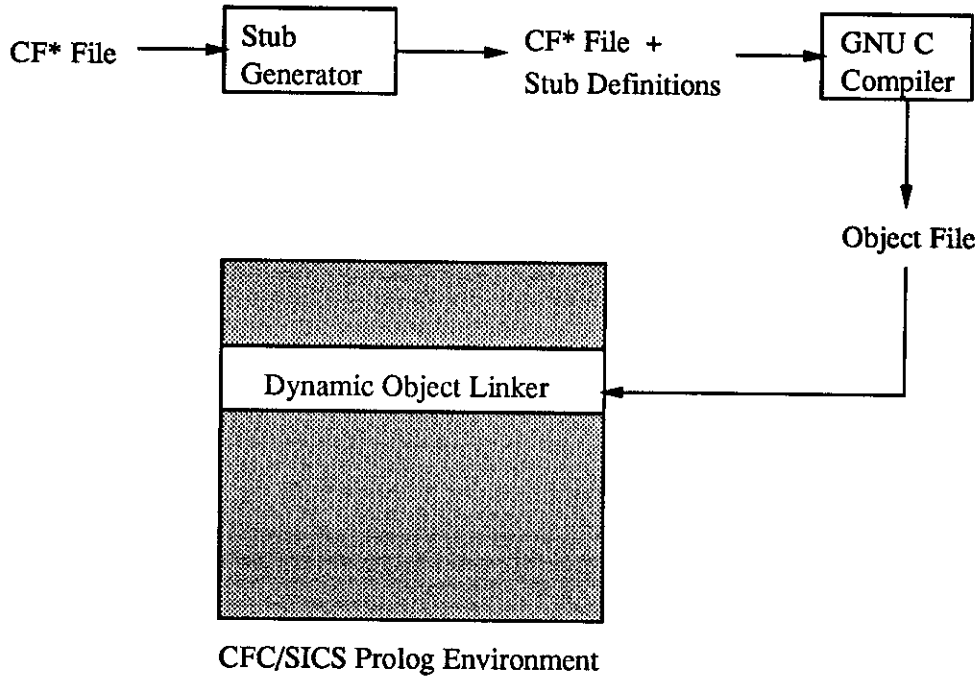

Figure 5.1: Linkage of CF* files in the CFC runtime environment.

modules are then dynamically loaded using SICStus Prolog facilities mentioned earlier.

In our present implementation, we provide a simple interface for calling compiled F* functions (i.e. CF* functions) from Prolog. A Prolog predicate is provided which translates terms from their Prolog representation to F* and vice versa. Also, a `reduce/2` predicate is defined which behaves like the Log(F) reduce.

Based on these primitives, a we have built a simple F* shell. A sample session which illustrates the usage of the F* shell is provided in appendix A.

CHAPTER 6

F* Programming and Extensions

6.1 Unification

As was discussed in Section 1, restriction (c) which limits the depth of unification in F* rules has been relaxed in our implementation. Even though this restriction was used in [Nar 88] solely for the purpose of simplifying the theoretical analysis of F*, it remains intact in the Log(F) implementation.

The following is an example of a rule which violates restriction (c), but is accepted by CFC.

```
listOfList([[X|Y]|Z]) => true.
```

This rule is compiled into the following DFAM function:

```
listOfList(x_1)
[
        addr    a_1;
        data    d_2,d_1;
        dreduce(x_1,d_1)                    % reduce 1st arg to d_1
        test(islst(d_1), goto(listOfList0)) % is d_1 a list?
        a_1 = tolst(d_1)                    % a_1 points to head
        d_2 = dreduce(*a_1)                 % d_2 is the reduction
                                            %   of the list's head
        test(islst(d_2), goto(listOfList0)) % is d_2 a list?
        dreturn(true)                       % return true
listOfList0:
        dreturn(fail)                       % return fail
]
```

In addition, CFC provides a general unification function, called `eq`. This function is able to unify 2 complex terms which may contain function terms invocations, by recursively traversing and reducing the terms.[1]

6.2 Eager Reduction

From a programming perspective, one of the more useful extensions that was implemented in CFC has been the *eager* operator, "`=>`". By using the "`=>`" operator, programmers can force the reduction of expressions which would otherwise be evaluated lazily. Forcing early reduction of certain terms, may result in substantial performance improvements. For instance, given the following F* rule:

```
square(X) => times(X, X).
```

If the argument send to the `square` rule is a function term, it will be evaluated twice, inside the multiplication function. Therefore, to avoid this problem, we can rewrite the `square` rule as:

```
square(X) => times(=>(X), X).
```

In the above rule, the argument X is reduced before it is sent to the multiplication function. That is, the following DFAM code is generated:

```
square(x_1)
[
        data    d_2,d_1;

        dreduce(x_1,d_1);
```

[1] Note that Log(F) also provides "=" as a unification operator. Unfortunately, "=" does not properly handle terms with embedded function call terms.

```
        d_2 = times(d_1,d_1);
        dreturn(d_2);
]
```

The eager reduction operator can be used to reduce not only variables, but also any F* expression. Therefore, it can be used by the programmer to explicitly eliminate common subexpressions in F* rules.[2]

6.3 External Interface

In order for F* terms to be passed to external systems (e.g. Prolog, relational database, etc.), the function term invocations within those terms must be completely evaluated. That is, each term must be structurally traversed, and any function terms evaluated.

In the following F* program, the `totally_reduce` rule can be used to completely reduce any given F* term.

```
extern([univ/1, vinu/1]).

totally_reduce(X) => vinu(reduce_univ(univ(X))).

reduce_univ([X|L]) => [X | =>(reduce_list(L))].

reduce_list([]) => [].
reduce_list([X|L]) => [=>(totally_reduce(X)) | =>(reduce_list(L))].
```

The `univ` library function behaves exactly as the Prolog `=../2`. That is, it returns a list containing the principal functor of the term and its arguments. The `vinu`

[2]It is possible perform eager reductions in a similar manner in Bop. The programmer can force the reduction of any Bop term within the *Condition* component of a rule. For instance, the rule for square would be written in Bop as:

$$square(X) \Rightarrow times(Y,Y) : -X =\gg Y.$$

routine performs the inverse.

As an example, the effect of applying `totally_reduce` to the term `[1+1, c(2*2, 11-3)]` would be the list `[2, c(4, 8)]`.

6.4 Function Inlining

Presently, many often used operation such as `if/3`, `eq/2`, `and/2`, `or/2`, `+`, `-`, `*`, etc. are implemented as library routines. Although this implementation greatly simplifies the F* compiler, it is rather inefficient. The addition of a function inlining feature to the compiler can considerably improve the performance of compiled F* code.

CHAPTER 7

Performance Measurements

In this chapter we compare the performance of CFC with Log(F) and Prolog. Unlike the comparison of CFC with Log(F), comparing the performance of a F* system with Prolog is a rather dubious task. Several factors affect the usefulness of any such comparison:

- In F*, function arguments are "input-only". That is, these arguments are not modified by the callee. In Prolog, however, variables can become *bound* as result of being passed to a predicate, and subsequently become *unbound* upon backtracking. Keeping track of current variable bindings is one of the major overheads in any Prolog implementation.

- Since F* terms can be lazy function calls, before performing any operation on terms, we must perform a reduction. Additionally, in F*, any such reduction may result in multiple non-deterministic values. That is, unlike Prolog in which only predicates are non-deterministic, any F* term may be non-deterministic. This adds to the complexity of implementing the non-deterministic control structures of F*.

- As was stated previously, the CFC implementation of lazy function terms is less than optimal. Therefore, any performance comparisons which heavily utilize lazy function terms in CFC are bound to be unfavorable to CFC.

- Many optimizations such as tail recursion optimization, better register allocation, tail call detection, dead code removal, etc. have not been implemented in CFC.

- Basic operations such as `if/3`, `eq/2`, etc. are not directly supported by the CFC compiler and are implemented as library calls.

In our performance comparisons we have tried to sidestep many of the issues mentioned above. The following sections provide results from several benchmarks which measure various aspects of CFC's performance. All measurements were performed on a Sun-3/260 with 8MB of main memory. Also, version 0.6 of the SICStus Prolog interpreter was used for the execution of the Prolog benchmarks.

7.1 Reverse Benchmark

The following implementations of `revn`, in Prolog and F*, reverse a given list N times.

Prolog:

```
revn([], L, L, 1) :- !.
revn([], L1, L2, N) :-
        NN is N-1,
        revn(L1, [], L2, NN).
revn([X|L1], L2, L3, N) :-
        revn(L1, [X|L2], L3, N).
```

F*:

```
revn([], L, N) => if(eq(N,1), L, revn(L, [], N-1)).
revn([X|L1], L2, N) => revn(L1, [X|L2], N).
```

Results:

DFAM/C	FAM/CF*	Log(F)	Prolog	Hand-optimized DFAM/C
265	394	2925	360	126

The table shows performance results with time displayed in milliseconds. All measurements were performed on a list of 450 elements. Many different values were used for the iteration argument to **revn**. The results displayed above are for 30 iterations (the iteration count did not affect the relative performance of the entries).

The DFAM/C entry represents the benchmark for F* code compiled into C. We subsequently hand-optimized the generated C code and placed the resulting run times in the "Hand-optimized DFAM/C" column. The optimizations performed were tail recursion optimization, dead code removal, register optimization, and inlining of the system library calls (all of which can be performed by an optimizing compiler).

Even though the **revn** F* rules are deterministic, for the purposes of this comparison, the CFC compiler was instructed to generate non-deterministic code in the form of CF*. This benchmark appears as the FAM/CF* entry.

The **revn** benchmark is primarily a measure of the unification (pattern match-

ing) performance. The Prolog, DFAM/C, and FAM/CF* entries seem to have very similar performance. The hand optimized DFAM/C entry, however, is approximately 2 times faster. The performance gains mainly resulted from tail recursion optimization.

On this benchmark, Log(F) is approximately an order of magnitude slower than the other entries. Similar results have been observed on other benchmarks.

7.2 Call Benchmark

In the following benchmark we measured function call (or predicate invocation) performance. The following Prolog and F* programs where used in this comparison:

Prolog:

```
calls :- a0(1).

a0(X) :- a1(X), a1(X).
a1(X) :- a2(X), a2(X).
a2(X) :- a3(X), a3(X).
a3(X) :- a4(X), a4(X).
a4(X) :- a5(X), a5(X).
a5(X) :- a6(X), a6(X).
a6(X) :- a7(X), a7(X).
a7(X) :- a8(X), a8(X).
a8(X) :- a9(X), a9(X).
a9(X) :- a10(X), a10(X).
a10(X) :- a11(X), a11(X).
a11(X) :- a12(X), a12(X).
a12(X) :- a13(X), a13(X).
a13(X) :- a14(X), a14(X).
a14(X) :- a15(X), a15(X).
a15(X) :- a16(X), a16(X).
a16(1) :- !.
```

F*:

```
calls => a0(1).

a0(N) => a1(=>(a1(N))).
a1(N) => a2(=>(a2(N))).
a2(N) => a3(=>(a3(N))).
a3(N) => a4(=>(a4(N))).
a4(N) => a5(=>(a5(N))).
a5(N) => a6(=>(a6(N))).
a6(N) => a7(=>(a7(N))).
a7(N) => a8(=>(a8(N))).
a8(N) => a9(=>(a9(N))).
a9(N) => a10(=>(a10(N))).
a10(N) => a11(=>(a11(N))).
a11(N) => a12(=>(a12(N))).
a12(N) => a13(=>(a13(N))).
a13(N) => a14(=>(a14(N))).
a14(N) => a15(=>(a15(N))).
a15(N) => a16(=>(a16(N))).
a16(1) => 1.
```

Results:

DFAM/C	FAM/CF*	Log(F)	Prolog
580	1610	8660	2400

The table above shows the overhead resulting from the usage of CF* calling mechanism over that of C.

7.3 Deep Backtracking

To compare the performance of DFAM/CF* non-deterministic function call mechanism, the following Prolog and F* programs where benchmarked.

Prolog:

```
do_fail :- fail.

back(0).
back(N) :- x(V), do_fail.
back(N) :- NN is N-1, back(NN).

%%
%% x/1 provides 210 choice points
%%
x(1). x(2). x(3). x(4). x(5). x(6). x(7). x(8). x(9). x(0).
....
x(1). x(2). x(3). x(4). x(5). x(6). x(7). x(8). x(9). x(0).
```

F*:

```
do_fail(X) => fail.

back(0) => true.
back(N) => do_fail(=>(x)).
back(N) => back(N-1).

%%
%% x can be reduced to 210 values.
%%
x=>1. x=>2. x=>3. x=>4. x=>5. x=>6. x=>7. x=>8. x=>9. x=>0.
...
x=>1. x=>2. x=>3. x=>4. x=>5. x=>6. x=>7. x=>8. x=>9. x=>0.
```

Results:

N	FAM/CF*	Log(F)	Prolog
50	120	800	419
100	210	1600	850
200	450	3200	1690

The table shows the run times for the various systems, using different values of

N. This benchmark is useful in the measurement of the systems' *deep backtracking* performance.

7.4 Shallow Backtracking

The following Prolog and F* programs were used to measure the performance of *shallow backtracking.*

Prolog:

```
do_fail :- fail.

back(0).
back(N) :- x(1,2), do_fail.
back(N) :- NN is N-1, back(NN).

x(X,1). x(X,1). x(X,1). x(X,1). x(X,1).
...
x(X,1). x(X,1). x(X,1). x(X,1). x(X,1).
x(X,2).
```

F*:

```
do_fail(X) => fail.

back(0) => true.
back(N) => do_fail(=>(x(1,2))).
back(N) => back(=>(N-1)).

x(X,1)=>true. x(X,1)=>true. x(X,1)=>true. x(X,1)=>true.
...
x(X,1)=>true. x(X,1)=>true. x(X,1)=>true. x(X,1)=>true.
x(X,2)=>true.
```

Results:

N	FAM/CF*	Log(F)	Prolog
50	40	320	60
100	120	640	100
1000	1060	6440	1020

These measurement show that the CF* shallow backtracking performance is essentially equivalent to that of Prolog. The Log(F) implementation is usually slower by a factor of 6.

CHAPTER 8

Conclusions

The primary impetus for this work came from our desire to implement an efficient stream processing environment based on F*. Even though F* was originally envisioned as an extension to Prolog, we have extended it to a full-fledged general purpose programming language. The resulting implementation offers a simple and efficient environment which elegantly combines logic programming, rewriting, and lazy evaluation.

This thesis has presented a mechanism whereby deterministic F* programs can be compiled into an abstract machine language called DFAM. We have shown that that it is possible to directly translate DFAM programs into C. The resulting C code runs better than an order of magnitude faster than the Log(F) implementation of F*. It was also shown that with the aid of further compiler optimizations, compiled DF* programs can run as much as 3 times faster than their Prolog counterparts running under SICStus Prolog.

The more general F* Abstract Machine (FAM) has been described in detail. We have shown that non-deterministic F* programs can be readily compiled into FAM programs. Furthermore, CF* was introduced as a novel extension to the C programming language, providing a non-deterministic function call mechanism.

A simple of mapping of the FAM instruction set to CF* was described. It was further shown that CF* can be efficiently implemented on conventional computer architectures.

We have also explored the possibility of utilizing the C programming language as an intermediate language for the compiler backend. This approach allowed us to create a highly portable compiler which provides reasonable performance, without the need to develop a machine code generator. Of course, the quality of machine code generated depends substantially on the C compiler used as the backend. We were fortunate to be using the outstanding GNU Optimizing C Compiler as the backend for the CFC environment.

This thesis has also explored the possibilities of extending F* into a full-fledged general purpose programming language. Even though the present implementation of the CFC system relies heavily on the availability of the SICStus Prolog C library, we believe that creating a stand-alone version of the system is not a difficult task.

In our work with the CFC, we have experimented with many new concepts in the implementation of stream processing systems. We believe that the underlying design decisions are sound and that with the addition of the optimizations and extensions suggested throughout this thesis, the CFC can become a truly "industrial strength" stream processing environment.

Bibliography

[Barr 85] Barrett, R. and Ramsay, A. and Sloman, A., "POP-11: A Practical Language for Artificial Intelligence," *Ellis Horwood Limited,* 1985.

[Boyd 90] Boyd J.L. and Karam G.M., "Prolog in C," *SIGPLAN Notices,* Vol. 25, No. 7, July 1990.

[Bruy 86] Bruynooghe, M., "Compile Time Garbage Collection," *Report No. 43,* Dept. Computerwetenschappen, Katholieke Universiteit Leuven, April 1986.

[Card 85] Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," *Computing Surveys,* Vol. 17, No. 4, December 1985, pp. 471-522.

[Cloc 81] Clocksin, W.F. and Mellish, C.S., "Programming In Prolog, Second Edition," Springer Verlag, 1984.

[Cohen 85] Cohen, J., "Describing Prolog by its interpretation and compilation," *Communication of the ACM,* Vol. 28, No. 12, pp 1311-1324, December 1985.

[Debr 86] Debray S.K. and Warren D.S., "Detection and Optimization of Functional Computations in Prolog," *Proceeding of the Third International Conference on Logic Programming,* London, U.K., July 1986.

[Ders 85] Dershowitz, N. and Plaisted, D. A., "Logic Programming *cum* Applicative Programming," *Symposium on Logic Programming,* 85CH2205-3, pp. 54-66, 1985.

[Gabr 85] Gabriel, J. and Lindholm, T. and Lusk, E.L. and Overbreek R.A., "A Tutorial on the Warren Abstract Machine for Computational Logic," *TR ANL-8-84,* Argonne National Laboratory, Argonne, Illinois 60439.

[Hayn 86] Haynes, C.T., "Logic Continuations," *Proceeding of the Third International Conference on Logic Programming,* London, U.K., July 1986.

[Kahn 84] Kahn, K.M., "How to Implement Prolog on a LISP Machine," *Implementations of Prolog,* Ed. Campbell, Ellis Horwood, 1984.

[Kral 87] Krall, A., "Implementation of a High-Speed Prolog Interpreter," *Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques,* pp. 125-131, St. Paul, Minnesota, June 1987.

[Levi 87] Levi, G., Palamidessi, C., Bosco, P. G., Giovannetti, E. and Moiso, C., "A Complete Semantic Characterization of K-LEAF, A Logic Programming Language With Partial Functions," *IEEE Symposium on Logic Programming,* 87CH2472-9, pp. 318-327, 1987.

[Liv 88] Livezey. B.K., "The Aspen Distributed Stream Processing Environment," Masters Thesis, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

[Mell 85] Mellish, C.S., "Some Global Optimizations for a Prolog Compiler," *Journal of Logic Programming,* Vol. 2, No. 1, pp. 43-66, April 1985.

[Muntz 88] Muntz, R.R. and D.S. Parker, "Tangram: Project Overview," *TR CSD-880032,* UCLA Computer Science Dept., LA, CA, April 1988.

[Nar 88] Narain, S., "LOG(F): An Optimal Combination of Logic Programming, Rewrite Rules and Lazy Evaluation," Ph.D. Dissertation, UCLA Computer Science Dept., Los Angeles, CA 90024-1596, 1988.

[Newt 87] Newton, M.O., "A High Performance Implementation of Prolog," *TR 5234-86,* Caltech Computer Science Dept., Pasadena, CA, April 1987.

[Nils 83] Nilsson, Jorgen Fischer, "On the Compilation of a Domain-Based Prolog," *Information Processing,* R.E.A Mason (ed.), Elsevier Science Publishers B.V. (North-Holland), pp. 293-299, 1983.

[Parker 88a] Parker, D.S., and Muntz, R.R. and Chau, L., "The Tangram Stream Query Processing System," *TR CSD-880025,* UCLA Computer Science Dept., Los Angeles, CA, March 1988.

[Parker 88b] Parker, D.S. and Muntz, R.R., "A Theory of Directed Logic Programs and Streams," *TR CSD-880031,* UCLA Computer Science Dept., Los Angeles, CA, April 1988.

[Parker 92] Parker, D.S., "Bop 0.2 Manual," UCLA Computer Science Dept., Los Angeles, CA, 1992.

[Sriv 85] Srivastava, A. and Oxley, D., "An Integration of Logic and Functional Programming," *IEEE Symposium on Logic Programming,* 85CH2205-3, pp. 254-260, 1986.

[Stic 88] Stickel, M.E., "A Prolog Technology Theorem Prover: Implementation by an Extended Prolog Compiler," *Journal of Automated Reasoning,* Vol. 4, pp. 353-380, 1988.

[Tama 84] Tamaki, H., "Semantics of a Logic Programming Language With a Reducibility Predicate," *IEEE International Symposium on Logic Programming,* 84CH2007-3, pp. 259-264, 1984.

[Tsan 88] Tsang, C.P. and Lewins, N., "Embedding Logic Variables in Scheme," *TR 88/18*, Dept. of Computer Science, University of Western Australia, Nedlands, W.A. 6009, Australia, 1988.

[Turk 86] Turk, A.K., "Compiler Optimizations for the WAM," *Proceedings of the Third International Conference on Logic Programming,* London, U.K., July 1986.

[van E 87] van Emden, M. and Yukawa, K., "Logic Programming With Equations," *Journal of Logic Programming,* Vol. 4, pp. 265-288, 1987.

[Wand 80] Wand, M., "Continuation-based program transformation strategies," *Journal of the ACM,* Vol. 27, No. 1, pp 164-180.

[Warr 83] Warren, D.H.D., "An Abstract Prolog Instruction Set," *TR SRI-83-309,* SRI International, Menlo Park, CA, October 1983.

[Wein 88] Weiner, J.L. and Ramakrishnan, S., "A Piggy-back Compiler For Prolog," *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation,* pp. 288-296, Atlanta, Georgia, June 1988.

APPENDIX A

Sample Programs

In this appendix, we present some simple F* programs and sample sessions with the CFC's F* shell. In interacting with the F* shell, the user is given a `=>` prompt, to which the user types F* expressions. Expressions of the form `load(`*file*`)`, dynamically load a compiled F* file.

A.1 Math

The math library is a collection of C routines which implements F*'s arithmetic operations such as addition, subtraction, etc. The following is a sample session which illustrates the usage of this library:

```
=> load(math).  % load the math library
=> 2+2.
4 ?
=> 100/50.
2.0 ?
=> 1+2+3+4.
10 ?
=> (1+2)*(3+4).
21 ?
=> (2*2)-4.
0 ?
=> 100 mod 4.
0 ?
=> 20 mod 7.
6 ?
=>
```

A.2 Append

The following are the rules for the F* append program:

```
append([], L) => L.
append([X|L1], L2) => [X|append(L1,L2)].
```

A sample session which illustrates the usage of append follows:

```
=> load(append).
=> append([1,2], [3,4]).
[1|'LF'(8,[2],[3,4])] ?
=> load([terms, reduce]).        % load totally_reduce
=> totally_reduce(append([1,2], [3,4])).
[1,2,3,4] ?
=>
```

The first reduction of append results in a list where the head element is 1, and the tail is a function term equivalent to `append([2],[3,4])`. In the second invocation of append the result is send to the `totally_reduce` rule which was described in Section 6.3.

A.3 Member

The following are the rules to the member function. When passed a list, it returns all elements in the list, non-deterministically.

```
member([X|_]) => X.
member([_|L]) => member(L).
```

A sample session which illustrates the usage of member follows:

```
=> load(math).
=> load(member).
=> member([1,2,3]).
1 ? ;
2 ? ;
3 ? ;
fail
=> member([2,2+2]).
2 ? ;
4 ? ;
fail
=> member([member([1,2]), 3]).
1 ? ;
2 ? ;
3 ? ;
fail
=>
```

A.4 N-queens

The following is a F* program for finding solutions to the N-queens problem:

```
extern([(+)/2, (-)/2, abs/1, not/1, (eq)/2, if/3]).

queens(X) => safe(perm(X)).

perm([]) => [].
perm([U|V]) => insert(U,perm(V)).

insert(U,X) => [U|X].
insert(U,[A|B]) => [A|insert(U,B)].

safe([]) => [].
safe([U|V]) => [U|safe(nodiagonal(U,V,1))].

nodiagonal(U,[],N) => [].
nodiagonal(U,[A|B],N) =>
        if(noattack(U,A,N),[A|nodiagonal(U,B,N+1)],none).

noattack(U,A,N) => not(eq(abs(U-A),N)).
```

To run the program, the queens rule must be passes a list [1,2,...,N], where N is the size of the chess-board. A sample run for a 4x4 chess-board follows:

```
=> load([eq, logic, math]).
=> load(queens).
=> totally_reduce(queens([1,2,3,4])).
[2,4,1,3] ? ;
[3,1,4,2] ? ;
[3,1,4,2] ? ;
[2,4,1,3] ? ;
fail
=>
```

Therefore, there are a total of two unique solutions: [2,4,1,3] and [3,1,4,2].

A.5 Primes

The following F* program computes a list of primes using Eratosthenes' sieves method.

```
extern( [if/3, (eq)/2, (+)/2, (mod)/2] ).

primes => sieve(intfrom(2)).

intfrom(X) => [X|intfrom(X+1)].

sieve([U|V]) => [U | sieve(filter(U,V))].

filter(A,[]) => [].
filter(A,[U|V]) =>
        if(multiple(U,A),filter(A,V),[U|filter(A,V)]).

multiple(U,A) => eq(U mod A, 0).
```

In the following sample run, the reduce_n library routine is used to totally reduce the first 10 elements of the list returned by the primes rule.

```
=> load([eq, logic, math]).
=> load([terms, reduce]).
=> load(prime).
=> primes.
[2|'LF'(19,'LF'(20,2,'LF'(18,'LF'(3,2,1))))] ?
=> reduce_n(primes, 10).
[2,3,5,7,11,13,17,19,23,29] ?
=>
```