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

THE ASPEN DISTRIBUTED STREAM PROCESSING ENVIRONMENT

Brian Kevin Livezey

December 1988
CSD-880102

UNIVERSITY OF CALIFORNIA

Los Angeles

The ASPEN Distributed Stream Processing Environment

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

by

Brian Kevin Livezey

1988

The thesis of Brian Kevin Livezey is approved.

RBagrodia

Rajive Bagrodia

[illegible]

D. Stott Parker

[illegible]

Richard R. Muntz, Committee Chair

University of California, Los Angeles

1988

TABLE OF CONTENTS

page

LIST OF FIGURES

page

ACKNOWLEDGEMENTS

I would like to thank Dick Muntz and Stott Parker for their guidance. Many thanks to Jon Edwards for his help in implementing several parts of the system. I'd also like to thank the other members of the Tangram group who offered lots of advice and support.

ABSTRACT OF THE THESIS

The ASPEN Distributed

Stream Processing Environment

by

Brian Kevin Livezey

Master of Science in Computer Science

University of California, Los Angeles, 1988

Professor Richard R. Muntz, Chair

Stream processing is an ideal paradigm for data-intensive applications. The solutions to a rich and varied set of problems that are, at best, awkward to express in other paradigms, can be expressed elegantly within the stream processing paradigm. Furthermore, stream processing presents an execution model in which such problems can be solved efficiently.

This thesis describes ASPEN, a stream processing environment. A programming language called Log(F) is extended to make it an appropriate language for expressing stream processing programs. The thesis focuses on those extensions that provide support for concurrent processing and access to distributed data.

The approach is novel in that the programming model allows the determination of the granularity of concurrency to be separated from the actual coding of the program. The degree of concurrency to be exploited is not fixed by the program specification or by the underlying system. Simple annotations allow the programmer to specify varying degrees of concurrency. Increasing or decreasing the degree of concurrency exploited during execution does not require rewriting the entire program, but rather, simply re-annotating it.

Several examples are given to illustrate the varying types of concurrency inherent in programs written within the stream processing paradigm. Examples are given which demonstrate how programs may be annotated to exploit these varying types and degrees of concurrency. The implementation of ASPEN is also described.

CHAPTER 1

INTRODUCTION

As computer applications grow in number, complexity, and size, there is a growing need for mechanisms which provide access to distributed data and allow concurrent processing. These mechanisms should provide increased performance in a transparent manner. Declarative programs should remain declarative even after being augmented with these mechanisms.

Future systems will require very high bandwidth access to data. Data may be stored in a database, generated on demand, or generated in real time. However the data is stored or generated, programmers should be allowed to access it in an identical fashion. When the required bandwidth exceeds that provided by current I/O devices, distribution of data and processing can greatly improve performance.

While many applications are data-intensive and benefit from the distribution of data as described above, other applications are more computation-intensive. They can often be divided such that individual portions may be computed independently. The performance of such applications may be improved greatly through concurrent processing.

Access to distributed data and concurrent processing, can both be provided in a uniform manner through the use of *streams*.

1.1 Streams

Streams represent ordered collections of data which are accessed in a sequential fashion. Streams can be produced and operated upon in a manner that is either *lazy*, *eager*, or some combination of the two.

Lazy production of a stream amounts to the production of a single data element and a *stream continuation*, a method for producing subsequent elements. When a consumer requests another data element, the stream continuation is *reduced* to produce a new data element and a new stream continuation. Thus, programs can process very large (or even effectively infinite) streams of data without requiring large amounts of memory to store the entire stream.

Eager production of a stream occurs when successive elements are appended to the end of the stream by the producer. When consumers request more data, they either receive it immediately or suspend until the producer supplies it.

Generally, lazy evaluation is used in sequential computations so that consumption of buffer space is reduced. Eager evaluation is useful in distributed computations to exploit parallelism. Eager and lazy evaluation can be combined to provide finer control over the flow of data through a network of processes.

1.2 User-Specified Concurrency

The degree of concurrency that can be exploited during the execution of a program should not be fixed by the structure of the program itself. Nor should it be fixed by the underlying system or language in which the program is written. These factors should determine only the maximal degree of concurrency that can be exploited.

Automated optimization in a distributed environment is an open problem. Currently, the only realistic way to attain real concurrency is through the use of programmer annotations. Eventually compilers with the capability to determine the degree of parallelism that can be effectively exploited within a given application may be developed. The output of such compilers will be annotated programs. As such, a language and environment such as ASPEN which supports annotations for parallelism is certainly a necessary first step toward supporting automated optimization.

1.3 Purpose of Thesis

This thesis describes ASPEN, a stream processing environment in which programmers can easily specify the degree of concurrency that is to be exploited in the execution of their programs. Programmers use simple *annotations* to indicate how to properly combine eager and lazy stream evaluation, concurrent processing, and distributed database access in order to achieve high performance for data-intensive applications.

Existing implementations of concurrent stream processing systems [2, 6] require programmers to structure their programs so as to reflect the degree of concurrency that is to be exploited during execution. It is much more natural for programmers to write their programs first, and later insert annotations which indicate the concurrency to be exploited. These practices allow the same program to be executed in different configurations with only minor changes to the annotations, thus exhibiting different performance characteristics, but producing identical results. The execution characteristics of a program can be changed without altering the program itself. Only the annotations need to be changed.

As stated earlier, lazy evaluation within a single process is very desirable. Buffering requirements are significantly reduced and partial results can be produced much more quickly than if eager evaluation is used. However, strictly lazy evaluation prevents the exploitation of a very important form of concurrency. By using annotations, the programmer can define process boundaries and introduce eager evaluation in order to exploit this form of concurrency.

In this thesis, the types of concurrency attainable in a distributed stream processing system are described. Annotations that allow a programmer to indicate when extra processes should be used to exploit each of these types of concurrency are presented. The implementation of ASPEN, a stream processing environment which supports such annotations, is also described. A prototype system has yielded encouraging results.

This research is part of the Tangram project [10, 14] at UCLA[†] whose goal is to develop a Prolog-based distributed modeling environment which combines DBMS and KBMS technologies with a variety of modeling tools.

1.4 Thesis Plan

This thesis is organized as follows. Chapter 2 provides necessary background information including a characterization of streams and stream processing. The third chapter describes how the concurrency inherent in ASPEN programs can be exploited through the use of annotations. Chapter 4 describes the implementation of ASPEN. Chapter 5 relates this work to work done previously by other authors. Chapter 6 proposes areas of future research. Finally, Chapter 7 contains concluding remarks.

[†]Tangram is supported by DARPA contract F29601-87-C-0072.

CHAPTER 2

STREAM PROCESSING

This chapter contains some background information necessary for understanding the rest of the thesis. A characterization of streams and *transducers*, the operations on streams is presented. Log(F), a language for composing transducers is described. Finally, the three types of concurrency inherent to stream processing programs are described.

All of the code in this thesis is specified in Log(F) and Prolog. See [16] for an introduction to Prolog and logic programming.

2.1 Streams

Streams, at the highest level of abstraction, can be viewed merely as ordered sequences of data objects which are accessed in a sequential manner. Thus, they could be implemented as Prolog lists. However, implementing streams as lists severely limits the power inherent in stream processing. Performing successive transformations on a list produces intermediate lists and a complete copy of the transformed list must be stored for each step. This storage is expensive or intractable for large streams.

Such storage problems are avoided by providing an implementation which permits *lazy evaluation* within a single process. When lazy evaluation is used, an element of the stream is only produced when it is needed. Transformations can be corou- tined so that one element of the input stream moves through a succession of operators until it is completely transformed. Only then is the next element of the input stream requested. Thus, one can avoid storing intermediate streams and one can write programs that manipulate potentially infinite streams.

2.2 Log(F)

This section presents a brief overview Log(F) [11, 12], a rewrite rule language developed by Sanjai Narain at UCLA. Log(F) provides lazy evaluation. With the extensions described in the next chapter, Log(F) is an excellent base language for expressing ASPEN programs. Log(F) allows programmers to express computations with a very fine degree of potential parallelism where each transducer potentially represents a process. ASPEN, as described in the next section, allows programmers to annotate these programs to indicate the desired amount of concurrency.

The following example illustrates how one composes Log(F) rules and how they are translated into Prolog for interpretation by a standard Prolog engine. The rules below describe how to append two streams.

```
append([ ], C)    =>  C.
append([A | B], C)   =>   [A | append(B, C)].
```

Prolog list notation is used to represent streams. In the term, `[A | B]`, `A` represents the first element of the stream, and `B` represents the stream continuation.

Log(F) rules are easily translated into Prolog *reduce* rules. The corresponding reduce rules for the above Log(F) rules are:

```
reduce(append(A, B), E)   :-
     reduce(A, D),
     append(D, B) => C,
     reduce(C, E).

append([ ], C)   =>   C.
append([A | B], C)   =>   [A | append(B, C)].

reduce([ ], [ ]).
reduce([H | T], [H | T]).
```

The symbols `[]` and `[H|T]` have reflexive reductions; that is, they reduce to themselves. Such symbols are referred to as *constructor symbols.*

When executed by a standard Prolog engine, reduce rules can be made to behave in a lazy fashion. If **`append([a,b,c], [d,e])`** were reduced once, the result **`[a|append([b,c], [d,e])]`** would be obtained. Further reduction of the tail would yield the result **`[b|append([c], [d,e])]`**. Thus, one can see that the computation is demand-driven; that is, no result is computed until it is needed.

2.3 Transducers

The elementary unit of computation in a stream-based language is the *transducer*. Abelson and Sussman [1] recognize four basic forms of transducers: enumerators, mappings, filters, and accumulators.

Enumerators take zero or more parameters and generate a stream of output. An example of an enumerator is a transducer which generates a list of all integers greater than some **N** given as input.

```
intsfrom(N)   =>   [N | intsfrom(N + 1)].
```

Note that this transducer produces an infinite stream. Another example of an enumerator is a relation stored in a database. The relation is viewed as a stream of tuples.

In the example above, +/2 is an eager operator. By default, all operators in ASPEN are lazy, that is, they are not computed until their results are needed. Some operators, arithmetic operators in this case, are defined to be eager. There is nothing to be gained by delaying their evaluation.

A *mapping* takes as input one or more streams and performs some sort of transformation on its input to produce an output stream. An example of such a transducer is the one below, which takes as input a stream of numbers and produces as output a stream consisting of the squares of the numbers on the input stream.

```
sq([ ]) => [ ].
sq([A | B])  =>  [A*A | sq(B)].
```

The first rule, the boundary condition, specifies that the square of all of the numbers on an empty stream is simply an empty stream. An example mapping from the database domain is a transducer that performs projections. For each input tuple, a new output tuple containing only the projected fields is produced.

Filters reduce the amount of data on a stream. Below is a transducer which filters odd numbers from a stream of numbers, producing a stream consisting solely of even numbers.

```
even([ ])  =>  [ ].
even([A | B])  =>
     if(is_even(A), [A | even(B)], even(B)).
```

Here, **is_even**/1 is a transducer which rewrites to true if the argument is even and rewrites to false otherwise. **if**/3 rewrites to its second argument if the first argument rewrites to true and rewrites to its second argument otherwise. Note in the above example that if **A** is odd, **even([A|B])** rewrites to a term that must be further rewritten before an output can be produced. Rewritable terms are rewritten automatically until a *constructor symbol* is produced. Constructor symbols are terms which rewrite to themselves. Common constructor symbols are `|`, `[ ]`, and numbers.

Accumulators perform aggregate functions on streams. They take a stream as input and generally produce a stream consisting of a single output. The example below produces the sum of the numbers on its input stream.

```
sum(S)   =>  sum(S, 0).
sum([ ], PS)   =>   [PS].
sum([A | B], PS)   =>   sum(B, PS + A).
```

Aggregate functions are common database operations; sums, averages, etc. occur quite frequently and can be expressed easily as transducers.

Hybrid transducers are quite common. For example, the transducer below, which produces a running total for a stream of numbers, is a hybrid of a mapping and an accumulator.

```
running_total(S)   => running_total(S, 0).
running_total([ ], PT)   =>   [PT].
running_total([A | B], PT)   =>
     [PT+A | running_total(B, PT+A)].
```

Transducers which both filter their input and perform mappings are also common. The transducer below takes as input a stream of numbers and produces as output a stream consisting of the square of every even number on the input stream.

```
sq_even([ ])   =>   [ ].
sq_even([A | B])   =>
     if(is_even(A), [A*A | sq_even(B)], sq_even(B)).
```

This transducer could, in fact, be written as the composition of a filter and a mapping.

```
sq_even_comp(S)   =>   sq(even(S)).
```

The above example shows that multiple transducers (of the same or differing forms) can be easily composed to form more complex transducers as in other functional programming languages. This mode of programming is common in ASPEN. Programmers define elementary transducers or obtain existing transducers from libraries and use them to compose more complex transducers.

2.4 Concurrency in Stream Processing Languages

Three basic types of potential concurrency are inherent in stream processing programs. They are stream parallelism, concurrent reduction of arguments, and merge parallelism. In this section, each type of concurrency is described and examples in which they arise are given.

2.4.1 Stream Parallelism

Stream parallelism is equivalent to pipelining. The potential for stream parallelism arises in stream processing programs when transducers are nested. It is illustrated by the example from the previous section which took a stream of numbers, filtered out the odd numbers and produced a stream consisting of the squares of the even numbers. The transducer was specified as **`sq(even(S))`**, where **`S`** was a stream of numbers. This transducer can be represented as the pipeline composition of two transducers as in Figure 1.

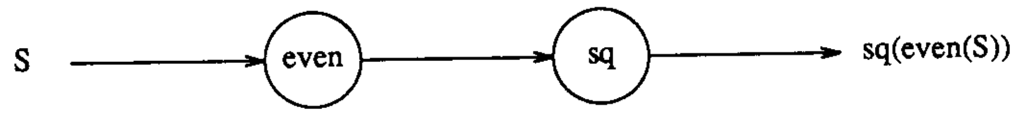

Figure 1 Pipeline Parallelism

Concurrent execution of both stages of the pipeline could potentially double the throughput of this transducer. Obviously, as the number of stages in the pipeline increases, the potential parallelism also increases.

Note that no parallelism is achieved if strictly lazy evaluation is used. However, if each of the transducers behave eagerly and run concurrently, parallelism is achieved. In the absence of multiple processes, lazy evaluation in this case yields an efficient use of storage as there is no need to store intermediate values.

2.4.2 Concurrent Reduction of Arguments

The potential for parallelism via the concurrent reduction of arguments arises in stream processing programs whenever transducers have multiple stream inputs. An example of this is a transducer which computes the sum of the squares of two streams, expressed as **`add(sq(A), sq(B))`**. Decomposition of this transducer yields the graph in Figure 2.

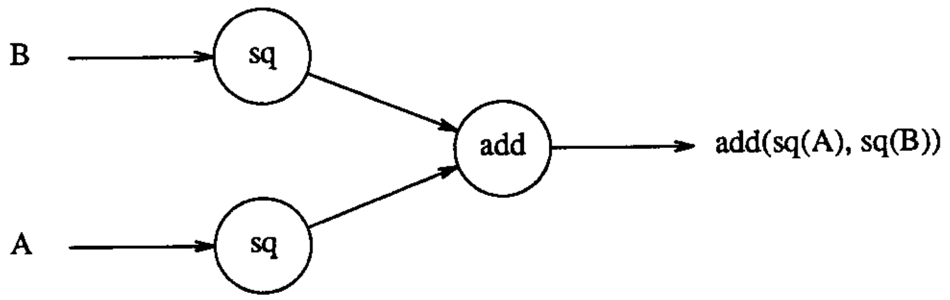

Figure 2 Concurrent Reduction of Arguments

Since the two inputs to **add** are independent, they can be produced concurrently.

In this case, concurrency can be achieved even when lazy evaluation is used. Multiple inputs to a transducer can be produced in parallel while remaining within the framework of lazy evaluation.

2.4.3 Merge Parallelism

Merge parallelism can be achieved if the task of producing a stream can be shared by multiple transducers. Consider a transducer which produces a stream that represents the relation, **R**, which is comprised of fragments, **R1**,...,**Rn**. This can be expressed as **`select([tuples(R1), ..., tuples(Rn)])`**. Decomposition of this function into its component transducers yields the graph in Figure 3. **`select`**/1 is a transducer which accepts as input a list of transducers whose outputs are to be interleaved in some unspecified manner to form one stream. **`tuples`**/1 is a transducer which produces a stream of output tuples corresponding to the relation name that is given as an argument.

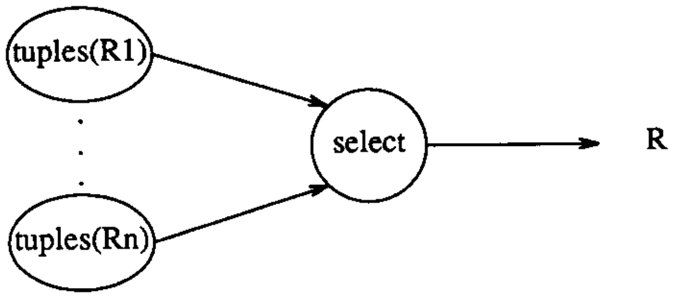

Figure 3 Merge Parallelism

CHAPTER 3
ASPEN PROGRAMMING TECHNIQUES

This chapter describes several of the constructs used when writing ASPEN programs. A single mechanism for achieving the basic types of concurrency described in the previous chapter is proposed. Additional mechanisms to improve the performance in cases in which this mechanism works, but yields poor performance are proposed.

3.1 Achieving Concurrency Through Annotations

Parallelism is achieved when different portions of a computation are performed concurrently on different processors. However, having every node in a transducer network represented by a different processor would result in too much overhead; the communication overhead would far outweigh the concurrency gained. Instead, a computation must be judiciously partitioned over the available processors. One must consider not only the potential concurrency in a given program, but also the overhead introduced by exploiting that concurrency. One must consider the computation costs of various portions of the program as well as the communication costs involved. Such factors are often determined by the nature of the input data as well as the structure of the program itself.

This section describes the primary concurrency annotations used by ASPEN programmers. The prefix annotation, **#**, is used by ASPEN programmers to indicate that it would be cost-effective to reduce the annotated term in parallel with the rest of the program. All three types of parallelism mentioned in the previous chapter can be achieved through the use of the **#** annotation.

Stream parallelism is achieved in the example of Figure 1 by annotating it as follows: `sq(#even(S))`. Such an annotation indicates that the filter `even` is to operate in a pipelined fashion with the mapping `sq`. Even integers are filtered out of the stream `S` on some remote site and streamed to the local site where they are squared as they arrive. Alternatively, the program could be annotated as `sq(even(S) @ ipswich)` to specify that reduction of the term `even(S)` is to take place on the site whose name is `ipswich`.

Concurrent reduction of arguments is achieved in a similar manner. The transducer `add(sq(A), sq(B))` is annotated as `add(#sq(A), #sq(B))` to indicate that the two input streams are to be produced concurrently.

Finally, merge parallelism is achieved in the example of Figure 3 by annotating it as follows: `select([#tuples(R1), ... , #tuples(Rn)])`. The exact interleaving of elements on the stream produced by `select`/1 is determined by the availability of data elements from each of the concurrent processes. The implementation of `select`/1 is discussed fully in Chapter 4.

Decisions about how to annotate ASPEN programs may be guided by several different factors. The programmer may have knowledge of the expected data and how it might affect computation costs. Some transductions may be known to be computationally expensive, while others are comparatively inexpensive. Many other factors may determine the best way to annotate a program, but regardless of how the program is annotated, the underlying program is not changed. Only the annotations are changed. Thus, the original specification of the transducer is the same regardless of whether it will be executed concurrently or sequentially.

ASPEN supports the annotation of rule invocations rather than the annotation of rule definitions for several reasons. Some invocations of a rule warrant the use of a separate process while other invocations of the same rule do not. The same rule may be invoked once to solve a problem which the programmer feels is likely to be expensive, while another invocation might be to solve a rather trivial problem. Secondly, since transducers are quite often defined in an iterative manner, annotation of rule definitions could cause excessive process creation. Recall the definition of **intsfrom**/1 from Chapter 2.

```
intsfrom(N)   =>   [N | intsfrom(N + 1)].
```

If the definition of **intsfrom**/1 were annotated, a new process would be created for every iteration, or for every output element produced. By annotating invocations, one can specify that the entire stream represented by **intsfrom**/1 is to be produced by a single process. Finally, the effect of annotating definitions can be achieved by annotating invocations. For example, if one really wanted **intsfrom**/1 to create a new process for each element it produces, one could easily achieve that behavior by re-writing **intsfrom**/1 as follows:

```
intsfrom(N)   =>   [N | #intsfrom(N + 1)].
```

Thus, annotating invocations is more general than annotating definitions.

3.2 Extending Log(F)

While the **#** annotation is sufficient for expressing all three types of concurrency, there are a number of important cases in which its structure is not entirely appropriate. In the following sections, those situations in which the **#** annotation alone is insufficient are described and alternatives which greatly improve performance are proposed. These language extensions are necessary for the efficient execution of sequential programs as well as concurrent programs.

3.2.1 Common Subexpressions

Pure functional programming languages allow programmers to construct only programs whose data flows are trees. Tree dataflows disallow the optimization of common subexpressions; if an expression is used in several places throughout a computation, it must be recomputed each time it is used. Allowing dataflows which are directed acyclic graphs (DAGs) rather than restricting the dataflows to be trees would permit optimizations such that these common subexpressions need only be computed once.

This section introduces a mechanism that allows common subexpressions to be optimized. This mechanism is available to the programmer so that he can specify when common subexpression is allowed. Common subexpression optimization could be done automatically by a compiler. However, programmers may not want this optimization to be applied to all common subexpressions. In some cases, the common subexpression may represent computations that are inteded to produce different results upon different invocations. They may be dependent upon side-effects or timing. In such cases, optimizing common subexpressions may alter the behavior expected by the programmer.

Henderson [5] describes the introduction of *local definitions* into a functional programming language to allow the optimization of common subexpressions. ASPEN provides a mechanism which appears to the programmer very similar to local definitions. However, local definitions in ASPEN programs must guarantee not only that the same initial value is assigned to all occurrences of a common subexpression;

they must also guarantee that whenever one occurrence of a common subexpression is reduced, the result of that reduction will be visible to all occurrences of the common subexpression. Thus, common subexpressions are never reduced more than once.

By using the infix operator, **where**, programmers can express transducers with optimized common subexpressions. The **where** operator is used to place conditions on terms. The right hand side of a rule defining a transducer will often be of the form: **Term where Condition**. Statements within **Condition** may bind one or more variables which occur in **Term**.

Consider, for example, the following transducer:

```
sq_plus_dbl(S)   =>  add(sq(S), dbl(S)).
```

Each instance of **S** is treated as a separate stream. So, each element of **S** is calculated twice, once for each consumer. This overhead could be quite significant if **S** represented a complex transduction. By rewriting the transducer as follows:

```
sq_plus_dbl(S)   =>  add(sq(R), dbl(R))
                 where
                 R <= S.
```

the overhead may be significantly reduced because the expression **S** is evaluated only once. Thus isolating common subexpressions in the **where** portion of a transducer guarantees that when any instance of the common subexpression is reduced, all instances will be reduced.

In order support the optimization of common subexpressions in distributed executions, ASPEN supports another use of the # annotation. By annotating a common subexpression with a # annotation, the programmer can specify remote reductions whose results are consumed by multiple processes. As with the in-line use of #, replacing # annotations with @/2 annotations in the **where** portion of a transducer allows the user to explicitly specify the site to be used for reduction.

With this use of the # annotation, the above transducer could be annotated for parallelism as follows:

```
sq_plus_dbl(S)   =>  add(#sq(R), #dbl(R))
        where
        R <= #S.
```

The generation of the input stream, R, the sq mapping, the dbl mapping, and the addition of the resulting streams can all take place in parallel. Here again, the expression S is evaluated only once, though the resulting stream will be replicated so that it may be consumed concurrently by the two transducers, sq and dbl.

3.2.2 Multiple Outputs

Pure functional languages make the expression of functions with multiple outputs very awkward. One method for allowing a transducer to produce multiple outputs would be to have the transducer produce structures as output. These structures would have one argument for each of the multiple outputs that the transducer is to produce. Two options are available here.

First, one could represent the output of a transducer as a stream of structures. This is acceptable when, any time one output is produced, all outputs are produced. Such is the case for div/2, which produces as its output, a stream of structures which contain the quotient and the remainder derived from the two input streams.

```
div([ ], [ ])  =>  [ ].
div([A | B], [C | D])  =>
     [t(A // C, A mod C) | div(B, D)].
```

Such a technique is awkward and inefficient when the outputs are produced at different rates. One of the elements of the structure must represent a term that has not yet been computed.

The second option, which deals more effectively with differing output rates, represents the output of a transducer as a single structure whose elements represent the multiple output streams of the transducer. Narain [12] proposed such a scheme for rewrite rules in Log(F). A rewrite rule, **partition**/4, which takes a single input stream and partitions it into two output streams based upon a pivot element, is expressed as follows.

```
partition(Pivot, [ ], L, R)  =>  t(L, R).
partition(Pivot, [A | B], L, R)   =>
     if(A =< Pivot,
          partition(Pivot, B, [A | L], R),
          partition(Pivot, B, L, [A | R])).
```

The output of such a rewrite rule might be consumed by a rewrite rule to quicksort a stream as follows.

```
quicksort([ ])  =>  [ ].
quicksort([H | T])  =>
     quicksort1(H, partition(H, T, [ ], [ ]).

quicksort1(A, t(L, R))  =>
     append(quicksort(L), [A | quicksort(R)]).
```

This option is unacceptable for several reasons. First, the syntax is very awkward; the third and fourth arguments to **partition**/4 are not intuitive and the need for **quicksort1**/2 is not intuitive. More importantly, **partition**/4 produces no output until the entire input stream has been consumed; such behavior which fits very poorly into the stream processing paradigm.

ASPEN supports an alternative approach in which all transducers produce a single stream of output. Elements of the stream are *tagged* to indicate the logical output stream to which they belong. Consumers merely filter out elements with the proper tag. The problem of representing and implementing transducers with multiple outputs is now reduced to that of representing and implementing common subexpression optimization.

To solve the multiple output problem in sequential executions, one makes use of the common subexpression optimization techniques described previously. The transducer for quicksorting a stream illustrates their use. In the following example, `o1`/1 and `o2`/1 are assumed to be constructor symbols and thus are not rewritten.

```
quicksort([ ])  =>  [ ].
quicksort([H | T])  =>
     append(quicksort(first(S)),
          [H | quicksort(second(S))])
     where
     S <= partition(T, H).

partition([ ], P)  =>  [ ].
partition([H | T], P)  =>
     if(H =< P,
     [o1(H) | partition(T, P)],
     [o2(H) | partition(T, P)]).
```

Note that **`partition`**/2 produces a single stream of output. Each element of that output stream is specified as either the first output (`o1`) or the second output (`o2`). This syntax is much more intuitive than that proposed by Narain. There is no need for the extra arguments required by Narain's implementation of **`partition`**/4. There is no need for the extra rule **`quicksort1`**/2. Most importantly, output elements are made available as soon as they are calculated, rather than being buffered until the entire output stream has been calculated.

The transducers **`first`**/1 and **`second`**/1 simply filter the appropriate output from the stream. They are defined below.

```
first([ ]) => [ ].
first([E | R]) =>
     if(E = o1(T), [T | first(R)], first(R)).

second([ ]) => [ ].
second([E | R]) =>
     if(E = o2(T), [T | second(R)], second(R)).
```

This set of filters is easily extended to allow functions with many outputs.

The quicksort transducer for distributed execution is shown below.

```
quicksort([ ])  =>  [ ].
quicksort([H | T])  =>
     append(#quicksort(first(S)),
          [H | #quicksort(second(S))])
     where
     S <= #partition(T, H).
```

The `partition` transducer need not be changed. The same filters that are used in the sequential case can also be used here.

In the distributed execution of `quicksort`/1, `partition`/2 produces two copies of its output stream. Both consumers receive all elements on the stream, regardless of the tag. Each stream is filtered as it arrives at its consumer. An obvious optimization is to force the filtration to take place at the producer, thus reducing the total amount of data that must be transmitted and buffered. For context-free filters like `first`/1 and `second`/1, this is a trivial optimization. For more complex filters (e.g. filters that maintain state), the optimization may be more difficult. Both cases are discussed in Chapter 6.

3.3 Increasing Merge Parallelism

Given the previously described mechanism for dealing effectively with transducers that have multiple outputs, another form of merge parallelism may be exploited. This form of merge parallelism arises when a node in a pipeline is replaced by a graph of processes which is capable of processing multiple elements of the stream concurrently. This situation is illustrated in Figure 4.

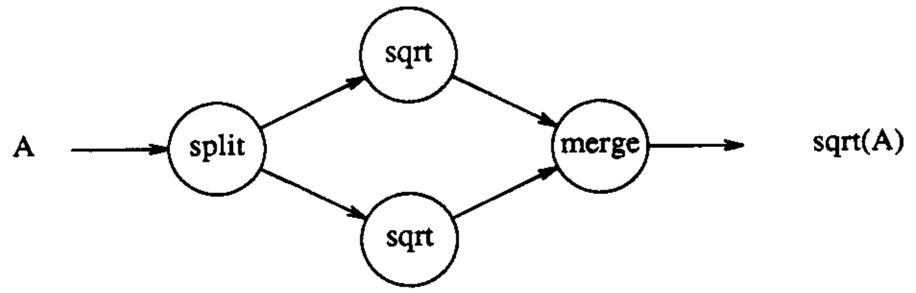

Figure 4 Merge Parallelism via Splitting

In this example, **A** is split into two streams, the **sqrt** mapping is applied to each, and the two streams are merged back into one stream. In this case, it is important that the stream ordering be maintained across the splitting and merging so that the result is guaranteed to be identical to that produced by a transducer which does not split the input stream. In other cases, the ordering constraint may be relaxed to increase potential concurrency.

Several possible methods for splitting and merging streams are enumerated below. Each has advantages and disadvantages both in terms of complexity and versatility.

If the split transducer and the merge transducer agree upon the method by which streams should be split, no mechanism is required to ensure proper synchronization of data. The transducers below illustrate such a simple method of splitting and merging streams.

```
split([ ]) => [ ].
split([X | Xs]) => [o1(X) | split2(Xs)].

split2([ ]) => [ ].
split2([X | Xs]) => [o2(X) | split(Xs)].

merge([ ], B) => B.
merge([A | As], B) => [A | merge(B, As)].
```

Note that the **`split`** transducer alternates production, first producing `o1(X)`, then producing `o2(X)`. The **`merge`** transducer consumes from its two input streams in a similar fashion, alternating between the two streams. This alternation is achieved by simply reversing the arguments on subsequent calls. As an illustration of how this set of transducers might be used, consider how one would annotate the program for the graph above. The filters, **`first`**/1 and **`second`**/1 were described previously.

```
t(A)   =>  merge(#sqrt(first(B)), #sqrt(second(B)))
      where
      B <= #split(A).
```

ordering is maintained automatically if all transducers between the split and merge operations perform one-to-one mappings.

A second method by which streams may be split and re-merged is to simply ignore ordering altogether. This method is similar to the use of **`select`**/1 described in section 3.1.3. The split transducer decides arbitrarily how to split the stream and the merge transducer simply accepts input from whatever stream has bindings available. This method perhaps offers better performance than the previous method since the merge transducer can consume whichever input is ready rather than having to wait for a specific one. However, this method is applicable only in cases where stream ordering is irrelevant.

If a standard ordering is maintained on the streams involved, another method may be employed. The **split** transducer takes as input a stream that is ordered according to some characteristic, e.g. ascending numeric order. The transducers that operate on the streams created by **split** must preserve order on the streams. Assume that the input to **split** is $[X_1, \ldots, X_n]$. For each $i \leq n$, X_i is sent on one of the two output streams where it is mapped by a sequence of transductions to $Y_{i,1}, \ldots, Y_{i,m_i}$, $m_i \geq 0$. For each $i, j \leq n$, $i \neq j$, if $X_i < X_j$, then $Y_{i,k} < Y_{j,l}$ for all $k \leq m_i$, $l \leq m_j$. This technique, while perhaps quite efficient for a limited set of applications, is useful only in cases where the sort criterion is maintained across all transductions which take place between the **split** and **merge** operations.

Another method of extracting merge parallelism is through cooperation between the **split** transducer and the **merge** transducer. In addition to producing streams of data, the **split** transducer also produces a stream of control information. This control stream is used by the **merge** transducer to assure that the streams are merged in the proper order. The ASPEN code for such a **merge** transducer is given below. The first argument to **merge** is the control stream. Each element, **E**, of the control stream indicates whether to accept input from the first stream (**E** = 1) or the second stream (**E** = 2).

```
merge([ ], X, Y)   =>   [ ].
merge([1 | Cs], X, Y)   =>   x_merge(Cs, X, Y).
merge([2 | Cs], X, Y)   =>   y_merge(Cs, X, Y).

x_merge(C, [X | Xs], Y)   =>   [X | merge(C, Xs, Y)].

y_merge(C, X, [Y | Ys])   =>   [Y | merge(C, X, Ys)].
```

One can now construct a transducer that splits its input stream based upon some characteristic of the elements. Such a method is very useful when intermediate transformations differ based upon the split characteristic. Each stream that is generated by the `split` transducer can be operated upon by a unique transducer, rather than trying to collapse the functionality of several transducers into one. Such a method is used in the Stream Machine [2] for an application in the field of oil exploration. Three streams of measurements are used to calculate the volumetric percentage of hydrocarbons in the rock formations at various depths in a borehole.

These measurements are lithology (the type of rock), electrical resistivity, and transit time (the time for a sound wave to propagate through the formation). The difference between the porosity of the rock and the percentage of water in the formation yields the volumetric percentage of hydrocarbons in the formation. The percentage of water is computed directly from the resistivity. The porosity is computed from the transit time, but different calculations are selected based upon the type of rock. It is for this calculation that the split/merge technique is applicable.

Program 1 shows a re-implementation of this example in ASPEN. The parameter `Rw` is the resistivity of water and is constant for each borehole. The `selectModel` transducer produces three output streams, `o1`, `o2`, and `c`. `o1` and `o2` are consumed by `sPorosity` and `lPorosity`, respectively. The stream `c` is used by `merge` to assure that the output streams of `sPorosity` and `lPorosity` are merged in the proper order.

This example introduces a new filter, `control`, which extracts control messages from a stream[†]. The transducer `case` is also introduced to allow the programmer to construct case statements as in other high-level programming languages. The code for each of these new transducers is given below.

[†]Note that the control stream could have been represented as the third output (`o3`) of `selectModel` and used the filter `third` rather than introducing a new filter. Often, however, the meaning of a program is clearer if the filters are given meaningful names.

```
hydrocarbons(Rock, Time, Resistivity, Rw) =>
     subtract(#porosity(Rock, Time),
          #water(Resistivity, Rock, Rw)).

porosity(Rock, Time) =>
     merge(control(R), #sPorosity(first(R)),
          #lPorosity(second(R)))
     where
     R <= #selectModel(Rock, Time).

selectModel([ ], [ ]) =>  [ ].
selectModel([Rock | Rs], [Time | Ts]) =>
     case(Rock, [
          sandstone: [o1(Time), c(1)
               | selectModel(Rs, Ts)],
          limestone: [o2(Time), c(2)
               | selectModel(Rs, Ts)]
          ]
     ).

water([ ], [ ], Rw) => [ ].
water([Resistivity | Rs], [Rock | R2s], Rw) =>
     case(Rock, [
          sandstone : [sqrt(0.8 * Rw/Resistivity)
               | water(Rs, R2s, Rw)]
          limestone : [sqrt(Rw/Resistivity)
               | water(Rs, R2s, Rw)]
          ]
     ).

lPorosity([ ]) => [ ].
lPorosity([Time | Ts]) =>
     [(Time - 47)/142 | lPorosity(Ts)].

sPorosity([ ]) => [ ].
sPorosity([Time | Ts]) =>
     [(5/8)*(Time - 55)/Time | sPorosity(Ts)].
```

Program 1 Oil Exploration Example

```
control([ ]) => [ ].
control([X | Xs]) =>
     if(X = c(E), [E | control(Xs)], control(Xs)).

case(Switch, [ ]) => [ ].
case(Switch, [Case : Term | S]) =>
     if(Switch == Case,
          Term,
          if(Switch == default,
               Term,
               case(Switch, S))).
```

The split/merge technique is very useful when the operations to be performed on the streams are complex and are intended to be executed in parallel. Perhaps even more significant is that this technique allows better software engineering of ASPEN programs. Note that the porosity models in the above example are defined independently of each other as transducers. Without using split/merge techniques, they would have had to be collapsed into one transducer or they would have had to be defined as operations on individual elements rather than streams, a severe restriction on their potential power as well as the degree of parallelism which may be exploited.

All of the above methods have the advantage of not requiring any modifications to the transducers that process the split streams. These intermediate transducers need not be aware that they are participating in a split/merge operation. The same code is used for the transducers whether they are participating in a split/merge operation or not. However, several of the options require that the intermediate transducers perform one-to-one mappings; otherwise, ordering cannot be maintained. An option which allows one-to-N mappings while still maintaining order on the stream introduces synchronization markers into the stream that is to be split. The Sync Model [8], a parallel execution model for logic programming, uses a similar technique to achieve parallelism in the all-solutions evaluation of logic programs.

A single synchronization marker precedes each term sent on the stream. Each synchronization marker on a stream has associated with it a number which is unique within the stream. These numbers are generated and introduced into the stream by the following transducer.

```
mark(X)    =>  mark(X, 0).
mark([ ], N)    =>   [ ].
mark([H | T], N)    =>   [sync(N), H | mark(T, N + 1)].
```

Any method of splitting the stream may be used as long as a data element and its associated synchronization marker are placed on the same output stream.

Transducers that encounter synchronization markers must be modified to echo them onto their output stream. In order to preserve determinism, "catch-all" rules which do not transform their input must be left unchanged. Such rules are characterized by having no formal parameters which force the reduction of their corresponding real parameter to a structure of the form `[A | B]`. Thus, only those transducers with one or more arguments of the form `[A | B]` need to be modified. Such rules must be modified so that, if **A** is a synchronization marker, it will be echoed onto the output stream. For each transducer, *f*/n, that participates in a split/merge operation, the following transformation must be performed.

1. if no arguments are of the form `[A | B]`, do nothing

2. if m arguments with indexes i_1 through i_m are of the form `[A | B]`:

   - rename all rules for f/n to f_m/n
   - for the first argument of the form `[A | B]`, add the following rule:

$$f(A_1,\ldots,A_{i_1-1},[X \mid Y],A_{i_1+1},\ldots,A_n) \Rightarrow$$
$$\text{if}(X == \text{sync}(SN),$$
$$[\text{sync}(SN) \mid f(A_1,\ldots,A_{i_1-1},Y,A_{i_1+1},\ldots,A_n)],$$
$$f(A_1,\ldots,A_{i_1-1},[X \mid Y],A_{i_1+1},\ldots,A_n)).$$

   - for the jth argument of the form `[A | B]`, add the following rule:

$$f_{j-1}(A_1,\ldots,A_{i_j-1},[X \mid Y],A_{i_j+1},\ldots,A_n) \Rightarrow$$
$$\text{if}(X == \text{sync}(SN),$$
$$[\text{sync}(SN) \mid f_{j-1}(A_1,\ldots,A_{i_j-1},Y,A_{i_j+1},\ldots,A_n)],$$
$$f_j(A_1,\ldots,A_{i_j-1},[X \mid Y],A_{i_j+1},\ldots,A_n)).$$

If each of the transducers participating in a split/merge operation is thus transformed, synchronization markers will be maintained in their proper order and determinism will be preserved.

Synchronization markers can now be used to merge the transformed streams in the proper order. The following transducer achieves this ordered merge.

```
merge([A | B], [C | D])  =>
    if(A == sync(M),
        if(C == sync(N),
            if(M < N,
          merge(B, [C | D]),
          merge([A | B], D)),
            [C | merge([A | B], D)]
        ),
        [A | merge(B, [C | D])]
    ).
```

Note that **merge** removes all synchronization markers from the stream and thus, disallows the nesting of split/merge pairs. This limitation can be eliminated through the use of labelled synchronization markers as shown below.

Labelled synchronization markers permit nesting while still maintaining order. The labelled forms of **mark** and **merge**, called **lMark** and **lMerge**, respectively are given below.

```
lMark(Label, X)   =>  lMark(X, Label, 0).
lMark([ ], Label, N)   =>  [ ].
lMark([H | T], Label, N)   =>
     [sync(Label, N), H | lMark(T, Label, N + 1)].

lMerge(Label, [ ], [ ])   =>  [ ].
lMerge(Label, [A | B], [C | D])   =>
    if(A == sync(Label, M),
        if(C == sync(Label, N),
            if(M < N,
                lMerge(Label, B, [C | D]),
                lMerge(Label, [A | B], D)),
            [C | lMerge(Label, [A | B], D)]
        ),
        [A | lMerge(Label, B, [C | D])]
    ).
```

Transducers that echo synchronization markers must be modified to echo this new type of synchronization marker. A common label must be agreed upon by **lMark/lMerge** pairs.

There are several options available when using split/merge techniques to exploit merge parallelism. The choice of which to use is dependent upon the application. One must consider whether ordering on the stream is important and, if so, whether the ordering can be exploited by the merge operation. Furthermore, one must consider whether all of the mappings are one-to-one and whether a simple method may be used or whether one of the more complex methods must be used.

3.5 Summary

This chapter has described how each of the three forms of concurrency inherent ASPEN programs may be exploited. Stream parallelism is achieved by pipelining nested transducers. Concurrent reduction of arguments allows multiple input streams to a transducer to be produced in parallel. Merge parallelism is achieved by having several transducers working in parallel to produce different portions of the same input stream to a transducer. Merge parallelism can be increased by splitting a single stream into multiple streams, allowing several transducers to operate on that stream in parallel and merging the resulting streams back together again. Most of the merge transducers given in this chapter merged two streams. Transducers which merge many streams may be expressed in an analogous manner. The next chapter describes the underlying system that has been implemented to support all of these types of parallelism.

CHAPTER 4

IMPLEMENTATION

This chapter describes the implementation of the ASPEN execution environment. The server model is described. The main topic of the chapter is the discussion of how worker processes provide service to clients. The extensions to the Prolog environment necessary to support this implementation are also detailed.

4.1 Server Model

This section discusses the only part of the implementation of ASPEN that is dependent upon aspects of Sun UNIX, the operating system on top of which ASPEN is currently implemented. If ASPEN were re-implemented on top of an operating system for which different cost assumptions were more appropriate and different features were available, the necessary changes would be isolated to the server model presented in this section.

Process creation in UNIX is a rather expensive operation. Since Prolog processes tend to be very large, their creation cost is especially high. It is unacceptable to pay the price of process creation every time one wishes to reduce a term remotely. Therefore, ASPEN provides pools of pre-existing processes which are capable of performing reductions. In order to further reduce the overhead of performing distributed computations, the ASPEN architecture facilitates efficient communication with these processes and efficient configuration of transducer networks.

Programmers construct transducer networks by sending service requests to remote machines. Programs that make such requests will be referred to as clients and processes on remote sites which accept those requests will be referred to as servers. Servers actually delegate work to members of a pool of processes called workers. This delegation of work is transparent to the client.

4.1.1 Servers

Each site that provides service allocates a service port to which clients can send service requests. A server process is allocated to wait for requests on that port. Servers are designed to have minimal impact upon performance. Their primary tasks are assigning work to worker processes and managing the size of the worker pool.

In order to reduce the overhead of communicating via a server, servers do not read incoming messages. They instead wait for an indication that a message has arrived and select a worker process to read the message and satisfy the request. This reduction in overhead is especially significant when satisfaction of the request involves relatively little processing as may be the case with some trivial database queries.

4.1.1.1 Forwarding Requests

All of the workers in a pool are descendants of the server that manages the pool. Thus, the server and all of the workers can share a file descriptor which identifies the service port. The server listens for connection requests on that file descriptor. When a request arrives, the server selects a worker to process that request. The server maintains the following semaphores in order to select an appropriate worker (one that is idle) and assure mutual exclusion on the service port. These semaphores are also accessible to all of the workers in the pool.

sem_idle	indicates the number of idle workers; incremented by workers when they finish processing a request; decremented by the server when a worker is selected for a task
busy[*i*]	indicates status of the *i*th worker; incremented by the server when the *i*th worker is selected; decremented by the *i*th worker when it finishes a task

4.1.1.2 Managing Pool Size

ASPEN is capable of adjusting to varying workloads. If the server realizes that all of its workers are busy and there are still connection requests queueing up, it can increase the size of its pool by creating another worker. Careful consideration should be put into the decision of whether to increase the pool size. If the currently allocated workers are suspended waiting for input, then it may indeed be beneficial to create another worker process that can satisfy requests while the other workers await input. If, however, requests are queueing up because all of the allocated workers are actively satisfying previous requests, creating additional workers may actually degrade performance by creating more contention for resources.

Likewise, if the server realizes that some of its workers are constantly idle, it can remove workers from the pool by killing processes and marking the appropriate semaphores to indicate that those processes are no longer available. This may be desirable if there is contention for swap space.

Decisions about pool size management can be quite complex. They can be based upon request queue length, average service time, average waiting time, throughput, or any combination of a host of other parameters. Making effective management decisions requires access to many performance parameters. The necessary instrumentation to measure such parameters is discussed in Chapter 6.

4.1.2 Workers

Workers are descendants of the server. As such, they are able to share a file descriptor for the service port. After creation, a worker's process image is overlayed with a Prolog image. The worker then consults a Prolog file which contains the code necessary for providing ASPEN service.

When a worker's semaphore is signaled by the server, it wakes up and accepts the connection request on its input port. Then it reads the reduction request from the established socket. The worker expects one of five message types. After reading the message and satisfying the associated request, worker *i* increments *sem_idle* and decrements *busy[i]*, making itself available to process further requests.

This section describes how worker processes handle the two most basic message types. Discussion of the other message types is deferred until sections 4.8 and 4.11. A message of the form `single(Term)` is received when a remote client wishes for the worker to reduce `Term` (an ASPEN expression, which, when reduced, will yield a stream of output) and send a single stream of results directly back to the client. A message of the form `multiple(Term, N)` arrives when a remote client wishes for the worker to reduce `Term` and produce output for `N` consumers.

4.1.2.1 Single Output

When the message `single(Term)` arrives at a worker on socket `S`, the goal `makeStream(Term, S)`, which is defined by the Prolog code in Program 2, is invoked. `makeStream(Term, Socket)` eagerly reduces `Term` and sends the results on `Socket`. When `Term` reduces to `[ ]`, a `writeStream` fails, perhaps because the client has closed the stream, or a reduction fails, `Socket` is closed and the worker becomes idle.

```
%  makeStream(Term, Stream)
%    the stream resulting from the reduction
%         of Term is sent on Stream
%    if, at any point, the reduction fails,
%         an error message is sent and
%         Stream is closed

makeStream(Term, Stream)   :-
     reduce(Term, ReducedTerm),
     !,
     sendTerm(ReducedTerm, Stream).
makeStream(Term, Stream)   :-
     writeStream(Stream, '$error'),
     closeStream(Stream),
     !.
makeStream(Term, Stream)   :-
     closeStream(Stream).

%  sendTerm(Term, Stream)
%    if Term = [ ], write '$end_of_stream' and
%         close the stream
%    if Term = [H | T], send H on Stream and
%         and continue reducing T and sending
%         the results on Stream

sendTerm([ ], Stream)   :-
     writeStream(Stream, '$end_of_stream'),
     closeStream(Stream).
sendTerm([H|T], Stream)   :-
     writeStream(Stream, H),
     makeStream(T, Stream).
```

Program 2 Creating Output Streams

4.1.2.2 Multiple Outputs

When the message **multiple(Term, N)** arrives at a worker on socket **S**, the following Prolog goal is invoked.

```
handle_request(multiple(Term, N), S) :-
     initialize(Term, N, S, Buffer, ServiceID),
     eagerLoop(Buffer, ServiceID).
```

The Prolog predicate **initialize/5** establishes network service, passes the port for that service back to the client and allocates a buffer structure to handle the reduction of the input term. The Prolog predicate `eagerLoop`/2 reduces the input term and sends results to all consumers until all consumers have either quit or received all of the results of reducing the input term. The Prolog specifications of `initialize`/5 and `eagerLoop`/2 are given in Program 3. `establishService`/2 returns a local port number and a service identifier to be used for accepting connections from remote clients. The goal `getHostName`/1 simply returns the name of the host on which it is executed.

Reduction of `Term` begins immediately. If connection requests from some of the consumers have not yet arrived, reduction results are buffered for them. Buffered results are disposed of when they have been sent to all consumers. The buffer manipulation predicates, `newClients`/3, `reduceTerm`/2, and `sendOutput`/2 are given in the specification of the worker process in Appendix C.

4.2 Specifying the Number of Consumers

The worker processes described in the previous section expect to be told how many processes will consume their output. As seen in Chapter 3, programmers merely annotate transducers to indicate that they are to be executed remotely, and place terms in the `where` specifications so that they may potentially produce output for multiple consumers. Programmers are not, however, expected to explicitly specify the number of consumers.

The ASPEN compiler translates rules of the form

```
LHS  =>  RHS  where  W.
```

```
%  initialize(Term, N, Stream, Buffer, ServiceID)
%    initialize a buffer structure for the
%         reduction of Term with N consumers
%    Stream is connected to initial client.
%    Buffer structures take the following form:
%         buffer(T, Tail, List, C, N)
%         - T is the term being reduced
%         - Tail is a pointer to a variable
%              in the buffer which represents
%              as yet unproduced elements
%         - List is the buffer with a variable
%              last entry pointed to by Tail
%         - C is the list of consumers and
%              their associated offsets
%              into the buffer
%         - N is the number of clients who
%              have not yet sent connection
%              requests
%    ServiceID is the address at which the worker
%         will receive connection requests
%         from clients

initialize(Term, N, S,
          buffer(Term, T, T, [ ], N), SID)   :-
     establishService(Port, SID),
     getHostName(Host),
     writeStream(S, streamDescriptor(Host, Port)),
     closeStream(S).


%  eagerLoop(Buffer, ServiceID)
%    until Buffer is empty:
%         - accept new clients on ServiceID
%         - reduce the term in Buffer
%         - send results to clients

eagerLoop(buffer([ ], [ ], [ ], [ ], 0), SID) :-
     !,
     shutdownService(SID).
eagerLoop(Buffer1, SID)   :-
     newClients(SID, Buffer1, Buffer2),
     reduceTerm(Buffer2, Buffer3),
     sendOutput(Buffer3, Buffer4),
     !,
     eagerLoop(Buffer4, SID).
```

Program 3 Supporting Multiple Consumers

to rules of the form

```
LHS  =>  RHS'  where  W'.
```

in such a way that `W'` and `RHS'` use only the primitives presented in this chapter. For each term of the form `T <= #S` in `W`, there is a goal of the form `remote(S, N, T)` in `W'` where `N` indicates the number of processes in `RHS` or in `W` in which `T` occurs. If `T` occurs multiple times within a single process in `RHS` or in `W`, common subexpression optimization is performed. Terms of the form `T <= Term @ Host` are treated in an analogous manner, being replaced by goals of the form `remoteSite(Host, Term, N, T)`. For each term of the form `T <= S` (where `S` is neither `#R` nor `R @ H`) in `W`, common subexpression optimization is performed. Whenever `T` occurs `N` times (`N` > 1) within a single process, it is replaced by a term of the form `common(CV, S, L)` where `CV` is an uninstantiated variable and `L` is a list of `N` uninstantiated variables. The significance of these variables is explained in the description of the implementation of `common`/3 which appears in section 4.6. If `N` = 1, `S` is substituted directly for `T`.

The following examples illustrate how the compiler modifies a user's code. Given the following input

```
sq_plus_dbl(S)   =>   add(sq(R), dbl(R))
    where
    R <= S.
```

the compiler would produce the following code.

```
sq_plus_dbl(S)   =>
    add(sq(common(T, S, [A, B])),
        dbl(common(T, S, [A, B]))).
```

As discussed in section 4.6, the sharing of the variable `T` assures that when one instance of the common subexpression is reduced, the result will be seen by all instances. If the above program were annotated for parallelism as follows:

```
sq_plus_dbl(S)   =>  add(#sq(R), #dbl(R))
    where
    R <= #S.
```

the following code would be produced by the compiler.

```
sq_plus_dbl(S)   =>  add(#sq(R), #dbl(R))
    where
    remote(S, 2, R).
```

The Prolog goal **`remote(S, 2, R)`** binds **`R`** to a descriptor for the stream produced by the process selected to reduce **`S`**. The implementation of **`remote`**/3 is discussed further in section 4.5.

Extensions to the compiler are described in Sections 4.4 and 4.6. The code for the ASPEN compiler is given in Appendix A.

4.3 Implementation of where/2

Many of the rules in this chapter use the transducer **`where`**/2. **`where`**/2 is defined by the following reduce rule.

```
reduce(where(Term, Condition), NewTerm) :-
    call(Condition),
    reduce(Term, NewTerm).
```

Here, **`Condition`** may be an arbitrary Prolog goal. The execution of **`Condition`** may bind logical variables which occur in **`Term`**, as with the variable **`R`** in the example of the previous section.

4.4 Cancelling Streams

Transducers do not always consume all of their input streams entirely. Frequently, transducers consume only part of a stream and then stop. Another common case is that a transducer never reduces one of its arguments. Unfortunately, these transducers never notify the producers of their input streams that they are no longer being consumed. The worker processes that produce multiple output streams expect that all of their consumers will eventually request input. If one or more of the consumers never request input, the worker will block forever.

Consider the following examples.

```
foo([found_the_term_I_wanted(X) | Xs]) => blah(X).
foo([last_term_I_care_about | Xs]) => [ ].
```

In both cases, it is known that **foo** will not consume any more of the input stream. The streams' producers must be informed of this fact. In order to do so, one must be able to express the fact that certain rewritings cause certain actions. ASPEN therefore supports the infix operator, **causing**/2, which has the following definition.

```
reduce(causing(Term, Action), NewTerm) :-
     reduce(Term, NewTerm),
     call(Action).
```

Here, **Action** is an arbitrary fragment of Prolog code. Note that **causing**/2 is similar to the **where**/2 transducer introduced earlier. The **Action** here is seen as a result of the reduction, while the **Condition** in **where**/2 is seen as a prerequisite for the reduction.

Using this new transducer, one can express the above examples as:

```
foo([found_the_term_I_wanted(X) | Xs]) =>    blah(X)
                              causing
                              cancel(Xs).

foo([last_term_I_care_about | Xs]) =>   [ ]
                              causing
                              cancel(Xs).
```

cancel/1 is a Prolog predicate which cancels any remote streams that occur within its argument. Its implementation is described in section 4.7.

Cases in which streams may be cancelled can be detected syntactically at compile time. Any stream which is mentioned in the left-hand side of an ASPEN rule but is not mentioned again in the right-hand side should be cancelled. Using this simple rule, unused streams can be detected as in the following transducer.

```
f(1, A, B, C)  =>  g(A).
f(2, A, B, C)  =>  h(A, B).
f(3, A, B, C)  =>  j(A, B, C).
```

This transducer is translated into the following form.

```
f(1, A, B, C)  =>  g(A)
     causing
     (cancel(B), cancel(C)).
f(2, A, B, C)  =>  h(A, B)
     causing
     cancel(C).
f(3, A, B, C)  =>  j(A, B, C).
```

The compiler can easily detect syntactically which streams must be cancelled for each rule of `f`/4. It is not necessary to determine which arguments are streams as cancellation of a non-stream argument has no effect. Transducers may be expressed without using `causing`/2 and `cancel`/1 and they will be compiled into transducers which cancel useless streams. The implementation of this compilation process is presented in Appendix A.

4.5 Implementation of Remote Annotations

Two basic types of distributed processing annotations were described in the previous chapter and earlier sections of this chapter. They were the # annotation for remote evaluation of a term with a single output, and those that translated into the Prolog goal `remote`/3 for remote evaluation of terms with multiple outputs. Each of these had a coinciding version which allowed the programmer to explicitly specify which remote site to use. The implementation of these annotations is described below.

When no site is specified, the **selectSite**/1 transducer is used to select one. **selectSite**/1 takes as input the term to reduce and rewrites to the name of the site that should be used to reduce the term. Currently, **selectSite**/1 simply selects sites in a round-robin fashion from a list of sites that are known to provide service. Suggestions for how **selectSite**/1 might make more informed choices of execution site (i.e. based upon loads, type of term to be reduced, stored relations mentioned in the term, etc.) are given in Chapter 6.

The host name selected by **selectSite**/1 is used to extend **#** to @/2 as shown below.

```
#T  =>  T @ selectSite(T).
```

The ASPEN rule for @/2 is:

```
Term @ Host =>
     if(Host = local,
          Term,
          remote_stream(Socket)
               where
               (reduce(Host, HostR),
                connectServer(HostR, Socket),
                writeStream(Socket, single(Term))
               )
     ).
```

The **remote**/3 goal is defined as follows, using the **selectSite**/1 transducer to decide which site to use for the remote reduction.

```
remote(T, N, S)  :-
     reduce(selectSite(T), H),
     remoteSite(H, T, N, S).
```

remoteSite/4 is defined by the following Prolog predicate.

```
remoteSite(local, Term, _, Term) :-  !.
remoteSite(Host, Term, N, StreamDescriptor) :-
     connectServer(Host, Socket),
     writeStream(Socket, multiple(Term, N)),
     readStream(Socket, StreamDescriptor),
     closeStream(Socket).
```

connectServer(Host, Socket) uses 4.3 BSD Unix socket calls to establish a connection, **Socket**, between the calling process and a server process on site, **Host**. **writeStream(Socket, Term)** simply writes **Term** to **Socket**.

With both **#** and **remote**/3, if the local site is chosen for the reduction, no new process is created and there is no overhead beyond the unannotated case.

The message **single(Term)** informs the server that **Term** is to be reduced and that the single consumer of the resulting stream will be the client who sent the message. The message **multiple(Term, N)** informs the server that there are to be **N** consumers of the stream produced by reducing **Term**.

When **@**/2 is used to perform a remote reduction, the following reduce rules are used to access the resulting stream.

```
reduce(remote_stream(Socket), Stream) :-
     readStream(Socket, Term),
     !,
     remoteStream(Socket, Term, Stream).

remoteStream(Socket, '$error', _) :-
     !,
     closeStream(Socket),
     fail.
remoteStream(Socket, '$end_of_stream', []) :-
     !,
     closeStream(Socket).
remoteStream(Socket, Term,
     [Term|remote_stream(Socket)]).
```

The programmer is provided with a stream of bindings that is accessed exactly as if it were produced locally. Successive reductions of the stream yield successive elements without requiring that the programmer know that the stream is produced remotely.

`remoteSite(Host, Term, N, S)` produces a *stream descriptor*, `S`, which identifies the stream which will contain the results of reducing `Term`. The client specifies the parameter `N` to indicate to the server that the stream will be shared by `N` consumers. This sharing is achieved by having the client pass the stream descriptor to all of the other consumers. The following reduce rule is used to further reduce stream descriptors.

```
reduce(streamDescriptor(Host, Port), Stream) :-
     connectStream(Port, Host, Socket),
     reduce(remote_stream(Socket), Stream).
```

Note that this reduce rule is used only for establishing the initial connection. Subsequent reductions of the stream use the rules defined above for reducing remote streams. In the above reduce rule, `connectStream(Port, Host, Socket)` establishes a connection, `Socket`, between the calling process and a process which is listening on the host-port pair, <`Host, Port`>.

4.6 Implementation of common/3

The optimization of common subexpressions within a single process requires the introduction of a new structure. The `common`/3 structure guarantees that each time one instance of of a common subexpression is reduced, the reduction will be seen by all other instances. To optimize a rule, `R`, which contains common subexpressions, occurrences of the common subexpression, say `X`, in a rule are replaced by `common(T, X, L)`. `L` is a list of variables with the number of elements equal to the number of clients and is used by `cancel`/1; its use is described in section 4.7. The significance of the logical variable `T` will be apparent after considering a simple example. As seen earlier, compilation of the rule

```
sq_plus_dbl(S)   =>
     add(sq(R), dbl(R))
     where
     R  <=  S.
```

yields a rule of the following form.

```
sq_plus_dbl(S)   =>
     add(sq(common(T, S, L)), dbl(common(T, S, L)).
```

When the first occurrence of `common(T, S, L)` is reduced, the logical variable `T` is bound to the result of the reduction. When the second occurrence of `common(T, S, L)` is reduced, the variable `T` is found to be bound, so its value is immediately returned as the result of the reduction.

`common/3` is implemented by the following Prolog code.

```
reduce(common(T, S, L), R) :-
     nonvar(T),
     !,
     R = T.
reduce(common(R, S, L), R) :-
     reduce(S, U),
     common(U, L, R).

common([A|B], L, [A|common(_, L, B)]) :- !.
common(T, L, T).
```

The first attempt to reduce `common/3` finds the first argument uninstantiated. Thus, the second reduce rule is used. The reduction is performed and the common logical variable, `T` in the above example, is bound to the result of the reduction. When the second instance of the `common/3` transducer is invoked, the logical variable is found to be bound and the first reduce rule is invoked returning the reduction result obtained previously.

Narain proposed a similar method for common subexpression optimization, in which every Log(F) rule is extended with an extra variable which serves the same purpose as the first argument to **common**/3. All rules are implemented in a similar manner to **common**/3 as described above; they first test to see if the extra variable is instantiated before actually performing the reduction. The implementation using **common**/3 has several advantages. First, the overhead of the extra variable and the extra rule (the one that checks to see whether the reduction has already been performed) is only incurred for those terms that occur as common subexpressions. Narain's method incurs the overhead for every occurrence of every term. The second advantage is that tracing ASPEN programs with **common**/3 is more intuitive. Finally, the **common**/3 approach provides support for stream cancellation by indicating how many copies of a common subexpression are in existence, and therefore, when remote streams can actually be closed. This point is discussed further in section 4.7.

A trace of the execution of **sq_plus_dbl**/1 through a few iterations demonstrates the behavior of **common**/3, If **sq_plus_dbl**/1 were called with **intsfrom(0)** as its input, the following behavior would be observed. Only those steps relevant to the understanding of **common**/3 are shown.

```
1. attempt reduction of sq(common(T, intsfrom(0), L))
        initially : T is unbound
        - the 2nd reduce rule for common/3 is chosen
          to reduce intsfrom(0) to [0 | intsfrom(1)]
        - thus, common(T, intsfrom(0), L) is reduced
          to [0 | common(A, intsfrom(1), L)],
          where A is the new logical variable which
          will be shared among the common subexpressions
        finally: T is bound to this result
2. attempt reduction of dbl(common(T, intsfrom(0), L))
        initially : T = [0 | common(A, intsfrom(1), L)]
                as a result of step 1 above
        - the 1st reduce rule for common/3 is chosen
          and the value of T is returned immediately
3. attempt reduction of sq(common(A, intsfrom(1), L))
        initially : A is unbound
        - the 2nd reduce rule for common/3 is chosen
          to reduce intsfrom(1) to [1 | intsfrom(2)]
        - thus, common(A, intsfrom(1), L) is reduced
          to [1 | common(B, intsfrom(2), L)]
        finally: A is bound to this result
4. attempt reduction of dbl(common(A, intsfrom(1), L))
        initially : A = [1 | common(B, intsfrom(2), L)]
                as a result of step 3 above
                            .
                            .
                            .
```

Each time **intsfrom**/1 is reduced (as opposed to using a previously reduced copy), a new logical variable (**A** in the above trace) is chosen to communicate reduction results between the instances of **intsfrom**/1.

4.7 Implementation of cancel/1

The Prolog predicate **cancel(S)** is responsible for cancelling remote streams mentioned in **S**. However, care must be taken not to cancel streams that are shared by other transducers within the same process. Such streams are indicated by the **common**/3 annotation and are thus easily detected. **cancel**/1 decomposes its input term looking for streams that require cancellation. All terms and subterms are recursively decomposed until an atom, a variable, or one of the following types of terms is encountered on each decomposition. If a term of the form **`remote_stream(S)`** or **`remote_stream(S, N, M)`** is found, a **`cancel`** message must be sent on the socket **`S`**. Upon encountering a term of the form **`streamDescriptor(Host, Port)`**, a connection is established with the worker process waiting at <**`Host, Port`**> and a **`cancel`** message is sent. This is necessary so that the worker will not end up waiting forever for a connection request. Another type of term that may be encountered is the **`common`**/3 term. When this term is encountered, its third argument is examined. If it contains more than one uninstantiated variable, one of the variables is instantiated to **`cancelled`** and decomposition stops. The existence of more than one uninstantiated variable indicates that another copy of this common term is still active. Note that instantiation of a variable in a common structure causes the same variable in all copies of the common structure to become instantiated. If there is only one uninstantiated variable in the third argument, then it is known that this is the last active copy of the common term and that it can be cancelled. This is done simply by continuing the decomposition on the second argument. The Prolog code for **`cancel`**/1 is given in Program 4.

4.8 Constrained Eagerness

Workers, by default, behave in an eager manner. That is, after they have reduced a term, they continue to reduce the stream continuation without waiting for the client to request such action. A worker stops producing output on its output stream(s) only when one of the following events arises:

- the transducer has produced the last element in the stream
- all consumers have closed their connections

```
% cancel(Term)
%    all streams mentioned in Term are closed
%         unless they are annotated as common
%         subexpressions and not all occurrences
%         are within Term

cancel(X)   :-
     var(X), !.
cancel(X)   :-
     atom(X), !.
cancel(remote_stream(S))   :- !,
     closeStream(S).
cancel(remote_stream(S, _, _))   :-  !,
     closeStream(S).
cancel(streamDescriptor(Host, Port))   :-  !,
     connectStream(Port, Host, Stream),
     closeStream(Stream).
cancel(common(_, S, L))   :-  !,
     mark_one(L),
     cancel_if_last(L, S).
cancel([ ])   :-  !.
cancel([A | B])   :-  !,
     cancel(A),
     cancel(B).
cancel(Term)   :-
     Term  =..  [_ | Args],
     cancel(Args).

mark_one([ ]).
mark_one([A | _])   :-
     var(A), !,
     A = cancelled.
mark_one([_ | B])   :-
     mark_one(B).

cancel_if_last(L, S)   :-
     all_marked(L), !,
     cancel(S).
cancel_if_last(_, _).

all_marked([A | B])   :-
     nonvar(A),
     all_marked(B).
all_marked([ ]).
```

Program 4 Cancel/1

When the normal stopping condition for a transducer has been reached (i.e. the stream continuation term has reduced to `[]`), the worker writes an `'$end_of_stream'` token to each output stream. The streams are then closed and the worker makes itself available for further requests.

When a consumer closes a stream that is produced by a worker, that action is detected by the worker. The worker closes the stream. If there are no more consumers associated with that worker, the worker informs the server that it is ready for more work. Otherwise, the worker merely removes the closed stream from its output set and continues producing output for the other consumers.

Though eager reduction of a term provides maximum parallelism, more efficient execution can often be achieved when the eagerness is bounded. Clients may decide, after seeing a portion of the stream, that they do not need to see any more. If the producer has eagerly produced the entire stream, much computation power may have been wasted. This situation will not arise if bindings are produced and consumed at the same rate. Very few bindings will ever be outstanding on the stream and when the consumer closes the stream, the producer will stop, having produced very few, if any, useless bindings.

If, on the other hand, data elements are produced at a greater rate than they are consumed, a large backlog of data elements may accumulate. Worse still, many processes may be created as the producer runs ahead of the consumer. If the consumer decides to stop consuming at some point before the end of the stream, the creation of these processes may have been entirely unnecessary. In such situations, much computation may be saved without sacrificing parallelism by producing the stream in a constrained fashion. Kahn and MacQueen [6] recognized this fact and introduced an *anticipation coefficient*, which was an integer that specified the maximum number of unconsumed items that may reside on a channel at any given time. This section presents a generalization of the anticipation coefficient which allows the programmer even more flexibility in controlling execution.

Specification of the production mode, *lazy(N, M)* (where N must be $\geq M$), indicates that a stream is to be produced in bursts. The first burst is to contain N elements, the maximum number that the consumer wishes to be outstanding at any time. After producing this first burst of bindings, the producer awaits a message from the consumer instructing the producer to continue. Such messages (called a **resume** messages) are sent by the consumer after the consumption of every Mth term. After receipt of such a message, the producer produces a burst of length M.

All modes of production along the continuum from completely lazy to completely eager can be achieved through the use of *lazy(N, M)*. By specifying *lazy(0, 1)*, completely lazy behavior can be achieved. A behavior in which bindings are produced on demand, N at time, can be achieved by specifying *lazy(N, N)*. A behavior in which the consumer is kept busy any time there are fewer than N bindings on the stream (equivalent to the anticipation coefficient above) can be achieved by specifying *lazy(N, 1)*. By specifying *lazy(∞, ∞)*, completely eager behavior is achieved. Note that specification of *lazy(∞, ∞)* is equivalent to specifying no constraint at all. The choice of values for N and M will be governed by the relative speeds of the producer and consumer and the amount of buffer space available between them as well as the expected behavior of the consumer. If it is expected that the consumer will terminate after consuming some relatively small number of bindings, then it may be desirable to keep the producer from running too far ahead and doing unnecessary work.

ASPEN provides a concurrency annotation which allows the specification of constrained reductions. The annotation `lazyRemote(N, M, Term)`, where `N` and `M` have the same meaning as described for *lazy(N, M)* is the constrained counterpart of `#`. Just as the `#` annotation may be translated by the compiler to the Prolog goal, `remote`/3, to handle common subexpression optimization, the `lazyRemote`/3 annotation can be extended to the Prolog goal, `lazyRemote`/5, for the same purpose. The `selectSite`/1 transducer is used to translate the `lazyRemote`/3 annotation into the `lazyRemoteSite`/4 annotation, as well as to translate the `lazyRemote`/5 goal into the `lazyRemoteSite`/6 goal. Program 5 shows the implementation of the `lazyRemoteSite`/4 annotation as well as the `lazyRemoteSite`/6 goal. In Program 5, reduction of `lazy_remote(Socket, N, M)` causes `N` terms to be read off of `Socket`. After `N` terms are read, a `resume` message is written. Then, reduction of `lazy_remote(Socket, M, M)` is requested.

At any time, a consumer can substitute a `cancel` message for a `resume` message. Such action would cause the stream to be closed. The worker will be deallocated if the consumer issuing the `cancel` message was the only consumer that the worker was serving. Otherwise, the worker merely removes the stream from its output set and continues producing output for consumers that sent `resume` messages rather than `cancel` messages. Note that the consumer could simply close the stream and achieve the same effect as that achieved by sending a `cancel` message.

Requesting the default production mode and simply closing the stream when the client does not wish to see any more input would achieve the same results as the scenario described above, but the performance would be much worse in those cases where the producer produces much more output than the consumer requires. While it

```
%  lazyRemoteSite(Host, N, M, Term)   =>  Stream
%    Stream is the result of rewriting Term on Host
%    Term is rewritten in lazy(N, M) fashion;
%         initially N results are produced;
%         thereafter, each time more results
%         are desired, a resume message is sent,
%         resulting in the production of a burst
%         of length M
%
lazyRemoteSite(Host, N, M, Term)   =>
          lazy_remote(Socket, N, M)
     where
      (reduce(Host, HostR),
       connectServer(HostR, Socket),
       writeStream(Socket, lazy(N, M, Term))
      ).

lazyRemoteSite(Host, N, M, Term, NC,
          lazy_remote(SD, N, M))   :-
      reduce(Host, HostR),
      connectServer(Host, Socket),
      writeStream(Socket, lazy(N, M, Term, NC)),
      readStream(Socket, SD),
      closeStream(Socket).

lazy_remote(Socket, N, M)   =>  Stream
      where
      (reduce(Socket, SocketR),
       readStream(Socket, Term),
       !,
       lazyRemote(Socket, Term, N, M, Stream)
      ).

lazyRemote(Socket, '$end_of_stream', N, M, [ ]) :-
      !,
      closeStream(Socket).
lazyRemote(Socket, Term, 1, M,
      [Term|lazy_remote(Socket, M, M)]) :-
          !,
          writeStream(Socket, resume).
lazyRemote(Socket, Term, N, M,
      [Term|lazy_remote(Socket, Nn, M)]) :-
          Nn is N - 1.
```

Program 5 Constrained Reductions

is true that a worker cannot do anything productive while waiting for a **resume** message, it is often best to have the worker idle so that other workers on the same site can perform useful reductions. The *lazy(M, N)* option allows the construction of more efficient transducer networks.

Upon receipt of a **lazy(N, M, Term)** message on socket **S**, the worker invokes the goal **lazyStream(N, M, Term, S)**. **lazyStream**/4 is defined by the Prolog code in Program 6.

```
% lazyStream(N, M, Term, Stream)
%    N is the number of solutions that should be
%         sent initially
%    M is the number of bindings that should be sent
%         in response to each 'resume' message
%    Term is the term to be reduced to produce the
%         next binding
%    Stream is the stream on which output
%         is to be sent

lazyStream(_, _, [], Stream)  :-
     !,
     writeStream(Stream, '$end_of_stream'),
     closeStream(Stream).
lazyStream(N, M, Term, Stream)  :-
     mSend(N, Term, Term2, Stream),
     readStream(Stream, resume),
     !,
     lazyStream(M, M, Term2, Stream).
lazyStream(_, _, _, Stream)  :-
     closeStream(Stream).
```

Program 6 Constrained Worker

Constrained workers make use of the predicate, **mSend(N, Term, Term2, Stream)**, which reduces **Term** **N** times, resulting in a new term, **Term2**. The results of these reductions are sent on **Stream**.

The constrained production of multiple outputs is similar. The code is specified in Appendix C.

4.9 Implementation of select/1

As discussed briefly in Chapter 3, the `select/1` transducer allows the programmer to exploit merge parallelism. `select/1` takes as its single argument a list of terms to be reduced. This list may contain an arbitrary mix of terms to be reduced locally and terms to be reduced remotely. This list is first partitioned into two lists, one containing local terms and one containing remote terms. Allocation of a worker process for each of the remote terms is requested. An auxiliary transducer `select/2` is then invoked. Its first argument is a list of stream descriptors for the allocated worker processes and its second argument is the list of local terms. `select/2` strives to return the first available binding.

The algorithm followed when reducing `select(Remote, Local)` to obtain the next result is outlined below.

1. If both `Local` and `Remote` are `[ ]`, return `[ ]`
2. If any remote streams have been closed, remove their identifiers from `Remote` forming `Remote'`
3. If a remote stream, has a data item, `D`, available, read it and return `[D | select(Remote', Local)]`
4. If no remote term is currently available, choose term, `T`, from `Local` and reduce it
5. If `Local = [ ]`, wait until a data item, `D`, becomes available on any remote stream, read it and return `[D | select(Remote', [ ])]`
6. If the result of the local reduction of `T` is `[ ]`, remove `T` from `Local` to form `Local'` and return the result of reducing `select(Remote', Local')`

7. If the result of reducing `T` is `[A | B]`, replace `T` in `Local` by `B` forming `Local'` and return `[A | select(Remote', Local')]`

A new Prolog built-in, **`selectStreams`**/8, which allows access to the Unix select system call is utilized here. It can be called in one of two modes, blocking or non-blocking. Blocking mode is used when there are no local reductions to be performed and the only alternative is to wait until one of the remote reductions returns a result. Non-blocking mode is used when there are local reductions to be performed; if a binding is available immediately on an input stream, it is read and returned. Otherwise, instead of waiting idly for a binding to arrive, a binding is produced locally by reducing one of the local terms. In Appendix B, the complete implementation of **`select`**/1 is given. Section 4.12 describes **`selectStreams`**/8 more fully as well as enumerating and describing other enhancements to the Prolog environment that have been made to support ASPEN.

4.10 Modules

All workers in a pool are endowed with the same capabilities and are therefore able to reduce the same set of terms. As database operations are expected to be very common, all workers are capable of solving database queries using a standard set of relational operators. However, workers may also be asked to reduce arbitrary terms which may include operations outside the standard set of relational operations. Reduction of such terms requires that the worker acquire new capabilities by loading a new module of code.

The existence of a module system which exhibits the following characteristics is assumed for the remainder of this chapter.

1. Unique names - A module name resolves to the same module of code, or an

identical copy of the module, regardless of which site the name is used on.

2. Encapsulation - Rules from other modules are visible only if they are explicitly imported by the current module. Thus, name conflicts are avoided and loading or unloading a module will not create conflicts with previously loaded modules.

3. Dynamic loading - When an imported predicate is called and the module that defines that predicate is already loaded, the predicate is simply invoked. If the module that defines the imported predicate has not been loaded yet, it is dynamically loaded and the predicate is invoked. Subsequent calls to that predicate (or any other imported predicate defined within that module) will find the module already loaded and the call will proceed immediately.

4. Unloading - Modules may be unloaded if they are no longer referenced. This is particularly useful for worker processes as they tend to be very long-lived and may be used by many different programmers for many different purposes over their lifetimes. Providing the ability to drop modules allows these worker processes to remain a manageable size.

5. Imports - All predicates that are imported by a module must be specified in the module definition. This restriction guarantees that an environment in a local client can be duplicated in a remote worker.

A module system for Log(F) which addresses some of these issues is described in [13].

Modules are either specified explicitly or implied by the client. To specify a module, the client simply includes the name of the module with the request for remote service as in the following example.

```
#f(X, Y) in Module.
```

where `Module` is a module in which `f`/2 is to be reduced.

If no module is specified, one of two assumptions is made. First, if the term to be reduced contains only database operations, no module is necessary as the worker already understands database operations. Second, if the term contains non-database operations, the current module of the client is assumed and the following reduction request is issued.

```
#f(X, Y) in C
where
active_module(C).
```

The call to `active_module`/1 is made on the local site and `C` is bound to the name of the currently active module. This name is passed to the worker process as a part of the request.

When a worker receives a request which specifies a module, the worker must load that module before attempting to satisfy the request. The module is unloaded only after the reduction is complete. Consider an example in which a worker receives the following request on socket `S`.

```
single(Term in Module).
```

The worker invokes the goal `makeStream(Term in Module, S)`. The `in`/2 transducer is defined as follows.

```
Term in Module  =>  Term
     causing
          ( drop_module(Module), module(Old) )
     where
          ( active_module(Old), new_module(Module) ).
```

4.11 Dedicated Workers

Constantly loading and unloading modules can result in serious performance degradation. For this reason, the concept of dedicated workers is introduced. A dedicated worker is endowed with a specified module and allocated to a specific client until that client terminates the dedicated service. Thus, if a client has a collection of terms that all require reduction within the same module, there is no need to reload the module for each of the terms to be reduced.

Clients request the service of a dedicated worker by issuing the following request.

```
dedicated_service(Socket, Term).
```

The logical variable `Socket` here insures that once the dedicated server is allocated, it is re-used rather than recreated on subsequent uses. This request is rewritten to `dedicated_host`/3,

```
dedicated_service(Socket, T) =>
     dedicated_host(selectSite(T), Socket, T).
```

whose implementation is shown in Program 7. The first two reduce rules are used when the request is to be satisfied locally. The behavior is different from performing standard reductions locally in that a module is loaded and remains loaded until the client terminates the dedicated service. While there is a context switch between modules on each call to `dedicated_service`, the module is dropped only after

```
%  dedicated_host(Host, Channel, Term)  =>  Stream
%    Stream is the result of the reduction of Term
%         by a dedicated server on Host
%    if Host = local, Channel indicates whether the
%         module has been loaded; if Channel is
%         unbound, then the module specified in
%         Term must be loaded before reduction
%    otherwise, Channel indicates whether a remote
%         server has been allocated to reduce
%         Term; if Channel is a variable,    then
%         a remote process is allocated and
%         Channel is instantiated to the socket
%         with which to communicate with that
%         process; subsequent calls simply send
%         reduction requests on Channel

reduce(dedicated_host(local, M1, Term in M2), S)  :-
     var(M1),
     !,
     active_module(Mold),
     module(M2),
     M1 = M2,
     reduce(Term, S),
     module(Mold).
reduce(dedicated_host(local, M, Term in M), S)  :-
     !,
     active_module(Mold),
     module(M),
     reduce(Term, S),
     module(Mold).
reduce(dedicated_host(Host, Socket, Term), S) :-
     isSocket(Socket),
     !,
     writeStream(Socket, Term),
     reduce(remote_stream(Socket), S).
reduce(dedicated_host(Host, Socket, Term), S) :-
     reduce(Host, HostR),
     connectServer(HostR, Socket),
     writeStream(Socket, dedicated(Term)),
     reduce(remote_stream(Socket), S).
```

Program 7 Dedicated Host

the dedicated service has been terminated.

The fourth reduce rule for **dedicated_host**/3 is used for the initial allocation of a remote dedicated server. Subsequent requests to a dedicated server use the third rule. Generally, the first call will specify a module in which to perform the reduction, i.e. **dedicated_service(Host, Socket, Term in Module)**. Subsequent reductions by that dedicated worker will use the same module.

The response stream coming back from the worker is essentially a stream of streams. Sub-streams are separated by **'$end_of_substream'** tokens. In order to properly process sub-streams, the following clause must be added to the definition of **remoteStream**/3 given in section 4.7.

```
remoteStream(Socket, '$end_of_substream', [ ]) :- !.
```

Thus, the consumer of the stream need not be aware that it was produced by a dedicated worker. It is treated just like any other stream.

When the client no longer requires the service of the worker, he relinquishes it by closing the socket. The programmer can use **dedicated_service**/2 for all interactions with the dedicated server. A compiler can recognize the last interaction with the dedicated server and modify it so that a **cancel** message is sent upon completion.

Dedicated service is intended to be used in a rather restricted class of applications in which invocations of the server will not overlap each other. That is, one invocation must complete before a subsequent request is issued.

Recursively constructed streams belong to the class of applications for which dedicated service is applicable. Consider the following transducer.

```
foo([ ]) => [ ].
foo([X | Xs]) => append(bar(X), foo(Xs)).
```

The reduction of **bar**/1 is assumed to require that the module **bar** first be loaded. This transducer could be annotated as follows.

```
foo([ ]) => [ ].
foo([X | Xs]) => append(
     dedicated_service(bar(X) in bar),
     foo(Xs)).
```

The above transducer is easily compiled into the following.

```
foo(A) => foo2(A, B).
foo2([ ], B) => [ ]
     causing
     cancel(B).
foo2([X | Xs], B) =>
     append(
          dedicated_service(B, bar(X) in bar),
          foo2(Xs, B)
     ).
```

The first invocation of **dedicated_service**/2 allocates a worker which loads **bar**, and binds **B**, the socket identifier. Subsequent invocations simply write the reduction request to the established socket.

Workers react to **dedicated(Term)** messages by reducing **Term** and then waiting on the input socket for the next request. When a **dedicated(Term in Module)** message arrives on socket **S**, the worker invokes the **dedicatedService(Term in Module, S)** goal given in Program 8. Dedicated workers react to requests in an identical manner to standard workers except that the worker is not deallocated after servicing the request. Instead, the worker awaits another request from the same client. Only after seeing a **cancel** message does the worker unload its module and deallocate itself.

If **cancel(M)** is called with **M** instantiated to a module name, it is assumed that the dedicated service was performed locally. Module **M** is simply dropped.

```
%  dedicatedService(Term, Socket)
%    Term is reduced by the dedicated server and
%         results are sent out on Socket
%    if Term specifies a module, that module
%         is loaded and remains loaded for
%         the duration of the dedicated service
%    the worker will continue to accept new terms
%         to be reduced in the originally
%         specified module until a cancel
%         message arrives; at this time, the
%         module will be dropped and the worker
%         will be deallocated

dedicatedService(cancel, Stream)  :-
     !,
     writeStream(Socket, '$end_of_stream'),
     closeStream(Socket).
dedicatedService(Term in Module, Socket)  :-
     !,
     active_module(M1),
     module(Module),
     dedicatedService(Term, Socket),
     drop_module(Module),
     module(M1).
dedicatedService(Term, Socket) :-
     reduce(Term, ReducedTerm),
     dedicatedRecurse(ReducedTerm, Socket).

dedicatedRecurse([], Socket)  :-
     writeStream(Socket, '$end_of_substream'),
     readStream(Socket, NextTerm),
     nextTerm(NextTerm, Socket).
dedicatedRecurse([H|T], Socket) :-
     writeStream(Socket, H),
     dedicatedService(T, Socket).

nextTerm(Term in Module, Socket)  :-
     !,
     dedicatedService(Term, Socket).
nextTerm(Term, Socket).
```

Program 8 Dedicated Server

4.12 Enhancements to the Prolog Environment

ASPEN servers are implemented as C processes, but in order to facilitate rapid prototyping and easy modification, the bulk of the other control is implemented in Prolog. In order to facilitate this, SICStus Prolog has been extended with some low-level primitives. These extensions fall into three categories, those for manipulating semaphores, those for manipulating streams, and those for manipulating Ingres relations.

The semaphore primitives allow the server and the workers to synchronize their activities.

`getSemGroup(+Num, -Group)` - allocate a block of **`Num`** semaphores, **`Group`** is instantiated to identify that block

`waitOnSemFree(+Semid, +Semnum)` - wait until the semaphore with index **`Semnum`** into the semaphore block, **`Semid`**, has been signaled

`waitOnSemNotFree(+Semid, +Semnum)` - wait until the semaphore with index **`Semnum`** into the semaphore block, **`Semid`**, has been decremented

`signalSemFree(+Semid, +Semnum)` - signal the semaphore with index **`Semnum`** into the semaphore block, **`Semid`**

`getSemValue(+Semid, +Semnum, -Semvalue)` - **`Semvalue`** is instantiated to the value of the semaphore with index **`Semnum`** into the semaphore block, **`Semid`**

The primitives necessary to support stream communication between processes are enumerated below.

`establishService(±Port, -ServiceID)` - service is established at the address, **`Port`**; **`ServiceID`** is the descriptor upon which connections may be accepted. If **`Port`** is uninstantiated when the call is made, **`Port`** is instantiated to a free port

`shutdownService(+Service)` - shut down the service identified by **`Service`**

`connectStream(+Port, +Host, -StreamID)` - a connection is established between the current process and a remote process at <**`Host,`**

Port>; **StreamID** identifies the created stream

acceptStream(+ServiceID, -StreamID) - a connection to **ServiceID** is awaited; **StreamID** is an identifier for subsequent reads and writes

closeStream(+StreamID) - the connection associated with **StreamID** is closed

setNoBlock(+StreamID) - writes to **StreamID** are to return immediately if there is no space available

readStream(+StreamID, -Message) - **Message** is the next Prolog term on **StreamID**

writeStream(+StreamID, +Message) - **Message** is written to **StreamID**; if there is no space, **writeStream**/2 blocks until space becomes available

writeStreamNB(+StreamID, +Message, -Return) - an attempt is made to write **Message** to **StreamID**; if there is space, the write succeeds and **Return** is instantiated to **success**; otherwise, the write returns immediately and **Return** is instantiated to **would_block**

selectStreams(-N, +RS, +WS, +ES, -RS', -WS', -ES', +M) - provides a general interface to Unix select system call; **RS**, **WS**, and **ES** are the streams which are to be checked to see if they are ready for reading, writing, or have an exceptional condition pending, respectively. **RS'**, **WS'** and **ES'** are the sets of streams that result from the above checks. **N** is the number of ready stream descriptors. **M** is the mode and is either **blocking** or **non_blocking** to indicate whether the call should block waiting for one of the descriptors to become ready or return immediately if no descriptors are ready.

The Prolog environment has also been extended to allow access to relational databases via the Ingres access methods. These extensions allow simple operations such as opening and closing relations, accessing tuples, and accessing schema information.

With this relatively modest set of enhancements, the quite powerful and varied functions presented in this thesis can be achieved.

CHAPTER 5

RELATED RESEARCH

This chapter compares ASPEN to related work done by others. First, ASPEN is compared with Flat Concurrent Prolog and Parlog, both parallel logic programming languages. ASPEN is also compared to the work of Kahn and MacQueen as well as the work done on the Stream Machine. This chapter is not intended to be a review of all related work. But, instead, it is intended to be a comparison with a representative set of related research efforts.

5.1 Flat Concurrent Prolog

Flat Concurrent Prolog [15] is a process-oriented logic programming language under development primarily at the Weizmann Institute of Science. Dataflow synchronization using read-only variables and guarded-command indeterminacy are the basic control mechanisms. In Flat Concurrent Prolog, a goal is viewed as a process, a conjunction as a network of processes, a shared variable as a communication channel, and the clauses of a logic program as rules for process behavior.

Stream processing is achieved in Flat Concurrent Prolog by repeatedly refining the instantiation of a shared variable. For instance, a list structure can be communicated by first instantiating some shared variable, say `X` to `[H | T]`, where `T` is unbound. `T` may be further bound to `[H2 | T2]`. Binding `T2` to `[ ]` signifies the end of the stream. The consumer instance of the stream is annotated to be read-only and the consumer blocks when it tries to read an unbound tail. This mechanism relies upon shared memory to allow access to the shared variables by all producers and consumers. By restricting the output streams of ASPEN transducers to contain only ground terms, the expense of dynamic synchronization, which is especially significant in loosely coupled systems, is avoided.

Sequential implementations of Flat Concurrent Prolog [9] require a mechanism to suspend processes which access uninstantiated read-only variables and a scheduler to decide which process is to be run based upon which read-only variables have been instantiated. In a sequential execution of an ASPEN fragment, no scheduler is required, stream continuations are used to produce exactly and only those values which are needed. There is no overhead of context switching or scheduling of processes whose outputs are not yet needed or may never be needed.

Flat Concurrent Prolog has been criticized for requiring the programmer to specify the correct synchronization via read-only annotations. CFL, a concurrent functional language has been proposed by Levy and Shapiro [7] as a user-level language for Flat Concurrent Prolog, relieving the programmer of this burden. Its implementation is based upon a source-to-source transformation of a CFL program into an equivalent Flat Concurrent Prolog program. CFL programs exhibit concurrency because of the parallel implementation of the resulting Flat Concurrent Prolog programs.

The current evaluation technique for CFL is eager evaluation. Calls and arguments to calls are evaluated in parallel. A lazy evaluation technique has been proposed, but it was deemed of limited use to combine lazy evaluation with unrestricted OR-parallelism. CFL provides no mechanism for the user to control concurrency.

Flat Concurrent Prolog requires modifications to unification. ASPEN does not require such changes and is fully Prolog-compatible. Thus, ASPEN can benefit from the current research into efficient Prolog implementations.

5.2 Parlog

Parlog [4] is a parallel logic programming language similar to Flat Concurrent Prolog. It provides synchronization by suspension on shared variables. Rather than using read-only annotations as in Flat Concurrent Prolog, Parlog predicates have mode declarations, indicating which arguments are to be used for input and which arguments are to be used for output. If an input argument is not bound when a predicate is called, the predicate will suspend. If an output argument is bound upon call, the predicate will fail.

Parlog supports stream AND parallelism, the concurrent execution of two predicates which share a variable whose value is communicated incrementally between the predicates. Stream AND parallelism is supported only for *single-solution* problems. That is, the shared variables must have the property of single-assignment; no binding of a shared variable, once made, will be changed.

Parlog provides the programmer with the ability to limit the parallelism which is to be exploited in the execution of his program. A conjunct of goals can be specified to execute concurrently by replacing `,`'s by `&`'s. For example, in the conjunct, g_1 `&` (g_2, g_3), g_1 must complete before the execution of the conjunct (g_2, g_3) begins. When the execution of g_1 completes, g_2 and g_3 are invoked in parallel.

Lazy evaluation can be achieved in Parlog by altering mode annotations. By reversing the roles of producer and consumer, one can have the consumer produce "boxes" for the producer to fill. The producer only runs when an empty box is available and the consumer only produces a box when it is ready for the next input. However, one cannot achieve lazy stream processing within a single process. While the Parlog conjunction g_1 & g_2 allows execution of g_1 and g_2 within a single process, no lazy evaluation is achieved; g_1 and g_2 are not coroutined.

No functional interface to Parlog has been proposed. Such an interface makes stream programming much more natural. Functional composition eliminates the need for mode declarations and allows implicit output specification.

5.3 Kahn and MacQueen

Kahn and MacQueen [6] introduced a model in which dynamically evolving networks of processes were used for the incremental generation and transformation of data. Buffered communication between processes is provided by uni-directional channels which behave like unbounded FIFO queues. The language is implemented in POP-2.

Processes are specified in process declarations of the following form:

```
Process <name> <parameter-list> ;
     <process body>
Endprocess
```

Parameters are one of three types: ordinary parameters, input ports, or output ports. Inputs are obtained by evaluating the expression **GET(A)** within the process body, where **A** is an input port specified in the parameter list. The procedure call **PUT(<expression>, B)** outputs the result of evaluating **expression** on the output port **B**.

Additional processes are created to evaluate subproblems through the use of reconfiguration instructions of the form:

```
doco  <body>  closeco
```

where **body** defines new processes and channels upon which they are to communicate.

Processes which generate a single output stream may be specified in a functional notation as follows:

```
Process <name> <parameter-list> => output;
     <process body>
Endprocess
```

Here, the parameter list contains only ordinary parameters and input ports. Using this notation, reconfiguration instructions can be expressed through functional composition and channels are created implicitly. Explicit input and output declarations and instructions are, however, still necessary.

Kahn and MacQueen's model seems to present two different programming paradigms to the programmer; a rather procedural paradigm for specifying processes and a more declarative paradigm for specifying configurations. ASPEN strives to present a more uniform declarative paradigm by not requiring programmers to explicitly manipulate processes.

Two modes of execution are presented, coroutine mode and parallel mode. In coroutine mode, execution is demand-driven as in sequential ASPEN. However, even in this mode separate processes are created by reconfiguration instructions. **GET** and **PUT** operations are still necessary to communicate terms between processes and a scheduler is needed to select the appropriate process for execution. These mechanisms and overheads are not incurred during sequential ASPEN executions. Demand-driven execution is the default behavior for sequential ASPEN and is achieved within a single process with no need for IPC or schedulers.

Parallel mode allows producers and consumers to execute in parallel as with ASPEN. It is not clear how one indicates which mode is to be used and whether these modes can be freely mixed to achieve the most efficient behavior.

5.4 Stream Machine

The Stream Machine [2], developed at Schlumberger, supports the development and execution of software for data acquisition under real-time constraints. It utilizes concurrently executing modules which communicate via streams in a data flow style. Modules in the Stream Machine are coarse-grained and are implemented as traditional sequential programs.

Access to values previously read from a stream is supported. However, for communication-intensive programs such behavior is not supported; previously read values cannot be accessed again and thus may be garbage collected. Values, once written to a stream, may not be retracted or altered.

Programmers must explicitly program the reading and writing of streams. Modules name input and output streams and manipulate them explicitly. These names are used in a separate segment of code which specifies how modules are to be linked together.

The programming style of ASPEN is much more natural. Reading and writing of streams is implicit. There is no need to name streams and use a separate language to link them together. Rather, inputs are represented in a functional style. That is, a stream is represented by the transducer that produces it.

When a stream is generated and consumed within the same process, no communication or scheduling should be required. Such is the case with ASPEN; values are communicated by variable bindings and the scheduling is expressed within the program itself. With the Stream Machine, however, even when modules reside within the same process, explicit I/O operations occur and a mechanism is required to select the appropriate module to execute next. This mechanism may take the form of a simple signaling mechanism or a more complex scheduler, neither of which should be required for a sequential evaluation.

Program 1 in Chapter 3 presented an example which had been translated from Stream Machine code to ASPEN code. The original Stream Machine program consists of six modules of Pascal code. The ASPEN code consisted of six transducers composed entirely in ASPEN. The Stream Machine code required an additional code segment to specify how the Pascal modules were to communicate. No such module is required for the ASPEN code; the relationships are specified in the transducers through functional composition.

The Stream Machine supports explicit manipulation of *piers*, the links between modules and streams that specify which element is to be read next. This allows the expression of *views* on streams. The authors give an example of a smoothing module that produces the average of three adjacent elements on the input stream. Such a transducer is easily expressed in ASPEN, requiring no extension to the underlying mechanism. The transducer to perform this operation in ASPEN is specified as follows.

```
smooth([])  =>  [].
smooth([H | T])  =>  smooth(T, H).

smooth([], A)  =>  [A].
smooth([H | T], A)  =>  smooth(T, A, H).

smooth([], A, B)  =>  [(A + B)/2, B].
smooth([H | T], A, B)  =>
     [(A + B + H)/3 | smooth(T, B, H)].
```

With some minor modifications to the ASPEN compiler, one could specify this transducer a bit more cleanly as follows.

```
smooth([])  =>  [].
smooth([A])  =>  [A].
smooth([A, B])  =>  [(A + B)/2, B].
smooth([X, Y, Z | R])  =>
     [(X + Y + Z) / 3 | smooth([Y, Z | R])].
```

Thus, views can be expressed in ASPEN without requiring the programmer to deal with any new constructs or operations.

CHAPTER 6

FUTURE WORK

This chapter presents some of the possible areas of future work that were discovered during the research, implementation, and writing of the thesis. Some of the ideas are large areas of research while others can be seen as incremental enhancements to the existing system.

6.1 Metaprograms

The current implementation requires that programmers actually modify the annotations of their source code in order to change the execution characteristics of their programs. While this is significantly better than requiring that they re-structure their programs, a much better approach would be to completely separate the specification of the program from the specification of the concurrency that is to be exploited.

By labelling annotations and specifying later what those labels mean, a programmer can alter the behavior of his program without changing the text of the code. For example, the programmer could write the following transducer.

```
sq_evens(S)  =>  sq(remote(sq_evens, even(S))).
```

In a separate *metaprogram*, the programmer could specify the meaning of the label.

The programmer might specify any one of the following.

```
label(sq_evens, remote).
label(sq_evens, local).
label(sq_evens, debug).
```

The following code could be used to interpret the annotated terms.

```
remote(Label, Term)  =>  do_remote(How, Term)
     where
     label(Label, How).

do_remote(local, T)  =>  T.
do_remote(remote, T)  =>  #T.
do_remote(debug, T)  =>  T
     causing ( notrace )
     where trace.
```

By altering the metaprogram only, the programmer can change the performance characteristics of his program. Labels for library transducers should be made known to the programmer so that the performance of these transducers may be modified as well.

Metaprograms for producers and consumers of multiple outputs are no more difficult to express or implement. Consider the following annotated transducer and its accompanying metaprogram.

```
sq_plus_dbl(S)  =>
     add(remote(sq, sq(R)), remote(dbl, dbl(R)))
     where
     S  <= remote(s, S).

label(s, remote).
label(sq, local).
label(dbl, local).
```

The transducer would be compiled to the following form.

```
sq_plus_dbl(S)   =>
    add(remote(sq,  sq(R)),  remote(dbl,  dbl(R)))
    where
    remote(s,  S,  2,  R).
```

The **remote**/4 call will bind **R** to a stream descriptor. Both consumers (**sq** and **dbl**) will reside on the local site. A possible optimization here would be to perform common subexpression optimization so that only one copy of the stream would have to be sent. However, the common subexpression optimization described earlier occurred at compile time. Here, it would have to occur dynamically at run-time.

6.2 Instrumentation

Ready availability of performance information about the system is essential for several reasons. As the design evolves, many decisions will be driven by the performance characteristics of the prototype system. Performance parameters must be available to the system itself so that it can dynamically reconfigure itself in response to varying demands. An example of such reconfiguration is the need to change the worker pool size on a given site. Such decisions might be based upon the length of the queue of requests at the server, the load on the processor, the average service time, or any combination of the above and other parameters.

Performance figures must be available to the programmer so that decisions can be made about how best to annotate a program for optimal performance. Providing programmers with information about, for instance, the rates at which each of the processes in a transducer network are producing output, would allow the programmer to locate bottlenecks and perhaps alleviate them by re-annotating his program. Measurements of processor time per term produced may help the programmer maximize concurrency while minimizing the overhead of transmitting terms.

6.3 Selecting Execution Sites

Information provided by the above-mentioned instrumentation can be used to automatically select a site upon which to reduce a term. Often, however, the choice of execution site is sensitive to the nature of the term. Through syntactic evaluation of the term (the term is not reduced), one can often determine what stored relations are used in the term. This information can be matched against a global data dictionary to determine the best site upon which to reduce the term.

6.4 Workspace Management

Most, if not all, Prolog systems suffer from design misfeatures which make them very difficult to use for large and/or long-lived applications. Among these problems are atom-table overflow and lack of modularity. These problems are described fully and solutions are proposed by Page in [13]

6.5 Interfaces

ASPEN is ideally suited to graphical programming. Conceptualizing an ASPEN program graphically where transducers are represented by nodes and streams are represented as arcs is very natural. Thus, providing a graphical programming interface would make programming much easier and increase programmer productivity.

Programmers must be allowed to view their programs at many different levels. At the highest level, an entire transducer network would be viewed as one node with input and output arcs. By "zooming in" on this node, the programmer should be able to see network of transducers which define it. The programmer should be able to expand any node in this network to any level of detail desired.

Programs should be constructed in a similar manner. Programmers construct small transducer networks and "zoom out," viewing these networks as single nodes and connecting them together with other nodes which themselves might represent previously constructed networks.

By viewing their programs in this way, programmers may be better able to make decisions about how to annotate them. Various portions of a given transducer network may be annotated to indicate that that portion represents a significant amount of computation and could likely benefit from concurrent evaluation. These annotations could be expressed by surrounding portions of the networks with boxes or by marking all of the nodes that comprise the annotated subgraph. Alternatively, one could label the output arc(s) of this subgraph and indicate in a metaprogram (as described above) what the labels mean.

6.6 Filters

As mentioned in Chapter 3, it would be more efficient to perform the filtration of streams that represent multiple outputs at the producing site rather than at the consuming site. Such an optimization is rather straightforward for such context-free filters as `first` and `second`. When a connection request is issued, the desired output type is specified to the producer. This output type specification is stored by the producer along with the list of client streams that is used when sending results. When an output is to be sent on a particular stream, it is first compared with the corresponding type specification. If it matches, it is sent.

Filters like `first` and `second` are relatively simple to perform on the producing site because they are context-free. Each output element either matches the filter or it does not. There is no dependence upon previous elements. Filters which are not context-free require that state be stored. For instance, the filter below accepts every third element on a stream.

```
every3rd(S)   =>  every3rd(S, 2).
every3rd([_ | S], 2)  =>  every3rd(S, 1).
every3rd([_ | S], 1)  =>  every3rd(S, 0).
every3rd([E | S], 0)  =>  [E | every3rd(S, 2)].
```

General filters of this form would be very difficult to implement as described above. Rather than storing a type specification and simply comparing the type of each output element to that specification, one would have to maintain a potentially large state specification and execute a potentially complex procedure for each element that is to be filtered. One would essentially end up implementing another ASPEN interpreter. Hence, one would have to allow the optimization of moving filters to the site of production only for context-free filters.

6.7 Debugger

ASPEN programs, even sequential ones, can be very difficult to debug. If a programmer uses the Prolog debugging facilities, he sees calls to predicates that he never mentioned in his program, most notably **reduce**/2. The appearance of such information serves to confuse the programmer who views his program as a collection of rewrite rules and not as a collection of **reduce**/2 rules. The programmer is likely to be further confused when **=>**/2 rules do appear but they are slightly modified versions of the ones that he specified.

Using the Prolog debugger, the programmer can place spypoints on either **reduce**/2 rules or on **=>**/2 rules. Since these are used as principal functors for all transducers, the user cannot place spypoints on individual transducers.

Annotated programs present further problems. The compilation process introduces Prolog goals that were not specified in the original program. These goals may confuse the programmer, both by their mere presence and by exposing details of the implementation that the programmer need not be aware of. Other issues also arise when debugging distributed programs. If a program is being traced and it creates another process, should that process be traced? If so, how should the information be communicated back to the programmer?

It seems clear that the Prolog debugger is inappropriate for debugging ASPEN programs, be they sequential or distributed. A special ASPEN debugger is essential. This debugger should allow the programmer to view the execution of a program as the program was written and not as it was implemented.

6.8 Automatic Optimization

The factors that determine how best to annotate an ASPEN program for concurrency are very similar to those that determine how best to execute a distributed query. One must balance computation costs against communication costs. Distributed queries are optimized by considering the sizes of the relations involved and the relative costs of the operations. The set of operations used in query processing is quite small, so the relative costs are quite well understood.

ASPEN programs involve a much broader range of operations. For this reason, completely unaided optimization would be impossible. However, if the programmer provides some advice, the techniques used to optimize distributed queries may be applied to the optimization of ASPEN queries. This advice would come in two forms.

First, information must be given about base relations. This information is identical to that required for distributed query optimization. One must know the number of tuples in the relation, the size of each attribute, and for each attribute **A**, the number of distinct values for **A** appearing in the relation. These parameters could be calculated for stored relations, but for relations that are generated in real-time (e.g. sensor readings), the programmer would have to be asked to estimate their values.

Secondly, the programmer would be asked to provide information about the transducers involved in the computation. Relevant parameters would include computation cost per tuple, the estimated ratio of the number of input tuples versus the number of output tuples, and the expected ratio of input tuple size to output tuple size. Using these parameters, weights can be assigned to each transducer.

Using the weightings of transducers and what is known about the base relations involved, it should be possible to optimally distribute processing over a given number of processors automatically. Thus, user annotations would no longer be necessary.

CHAPTER 7

CONCLUSIONS

Stream processing is a powerful programming paradigm. It allows a rich and varied set of programs to be expressed elegantly and solved in an efficient manner. Several interesting applications of stream processing are presented in [14] and [3].

This thesis has presented extensions to a programming language called Log(F) to make it more useful as a stream processing language. The most significant of these extensions are the annotations that allow programmers to exploit concurrency without rewriting their programs. A term may be annotated with the # annotation if the programmer wishes to indicate that the performance of his program could be improved by evaluating that term in parallel with the rest of the program. The @ annotation allows the programmer to explicitly specify a site upon which the reduction of a term is to take place. This can be extremely useful when the term to be reduced requires access to data which resides only at a specific site. The thesis has also presented annotations which may be used to make local definitions, allowing the programmer to express programs whose dataflows are DAGs as well as trees. Using these simple annotations, programmers can express programs which exploit three types of parallelism, stream parallelism, concurrent reduction of arguments, and merge parallelism, making ASPEN a powerful stream processing system.

The annotations presented in this thesis allow programmers to freely mix eager and lazy evaluation. Portions of an ASPEN program which run within a single process are evaluated in a lazy fashion. This is desirable for two reasons. First, there is no need to buffer intermediate results since each result is calculated only when it is needed. Second, the delay before the first result is produced is much lower than if eager evaluation were used. These factors are extremely significant for large, potentially infinite, streams. By introducing eagerness at process boundaries, concurrency can be exploited. Further annotations allow the programmer to constrain this eagerness in order to prevent one process from running too far ahead of another and requiring unbounded buffer space.

The annotations presented in this thesis are intended to be, to the greatest extent possible, semantics-free. That is, their introduction into a program changes the execution behavior of the program only, without changing the semantic meaning of the program. When annotations which explicitly specify an execution site are used to gain access to a particular database relation or fragment, there is, of course, a semantic impact on the program. Those annotations which do not explicitly specify an execution site can be taken as *hints*, to be either ignored or heeded during execution, at the discretion of the system. Since the annotations have no impact on the semantics of the program, there is significant potential for making program distribution decisions automatically, with no need for the programmer to annotate his program.

As ASPEN is used more extensively by the members of the Tangram research project, its strengths and weaknesses will no doubt become more apparent. We believe that the strengths will outnumber the weaknesses and that addressing the weaknesses will require only incremental changes to the implementation presented here.

APPENDIX A

ASPEN COMPILER

In this appendix, we show the code for the compiler described in Sections 4.2, 4.4, and 4.6. This compiler translates a program specified by the user into a Log(F) program with three very important properties. First, user-specified rules which do not consume all of their input streams are translated into a form that guarantees that unconsumed streams will be closed and their producers will not be made to suspend indefinitely. Second, if the programmer has specified a term as a candidate for local common subexpression optimization, that optimization is performed, here, at compile time. Occurrences of a common subexpression are tagged with a common variable so that when one occurrence is reduced, all occurrences see the effect. A cancellation list is also added to each occurrence so that the common term is only cancelled after all occurrences have been cancelled. Finally, requests for remote reduction of common subexpressions are extended to indicate the expected number of consumers.

```
%
%  dlc - ASPEN Compiler
%
%  ASPEN-to-Log(F) compiler to support ASPEN
%    - modifies ASPEN rules so that unused streams are
%         cancelled
%    - assures that producers of common subexpressions
%         will be informed as to the correct number
%         of consumers both for local and remote cases
%
%    called with a single filename or a list of files
%         - for each file <filename>.logf,
%              a file <filename>.dlc.logf is created
%         - for each file not of this form, <filename> is
%              extended to <filename>.dlc.logf
```

```
:- op(1150, xfx, (=>)).
:- op(730, xfy, where).
:- op(730, xfy, causing).
:- op(730, xfy, (<=)).
:- op(730, fy, #).
:- op(730, xfy, @).
:- op(725, yfx, in).
:- op(715, xfy, :).

:- ['/usr/local/lib/prolog/lib/sets'].
     % to define listtoset/2, memberchk/2, and subtract/3

:- ['/usr/local/lib/prolog/lib/lists'].
     % to define remove_dups/2 and append/3

dlc([]) :- !.
dlc([H|T]) :- !,
     csol(H),
     dlc(T).
dlc(F) :-
     dlc1(F).

dlc1(F) :-
     filenamePrefix(F, ".logf", Prefix),
     extendedFilename(Prefix, ".dlc.logf", EF),
     seeing(CurrentIn),
     see(F),
     telling(CurrentOut),
     tell(EF),

     repeat, read(T),
          ( T == end_of_file ->
               !, true ;
               ( once((
                    add_cancels(T, T2),
                    cso(T2, T3),
                    writeq(T3), write('.'), nl
                 )),
                 fail
               )),

     told,
     tell(CurrentOut),
     seen,
     see(CurrentIn).
```

```
%
%  add_cancels(+Term, -NewTerm)
%    for each variable, V, that occurs on the LHS of
%    Term, %  but does not occur on the RHS of Term, a
%    goal of the %  form, cancel(V), is added to the
%    'causing' portion of Term, resulting in the term
%    NewTerm.  If Term has no 'causing' portion, one
%    is added
%
add_cancels(T, T2) :-
     numbervars(T, 0, _),
     add_cancels1(T, T2).

add_cancels1((LHS => RHS), R2)  :-
     get_vars(LHS, LV),        % vars from LHS
     remove_dups(LV, LVs),
     rhs_vars(RHS, RV),        % vars from RHS
     remove_dups(RV, RVs),
     subtract(LVs, RVs, CVs), % LHS - RHS = {streams
                               %   that may need to be
                               %   cancelled}
     newRule(LHS, RHS, CVs, R2).

%
%  get_vars(+Term, -ListOfVariablesOccurringInTerm)
%
get_vars('$VAR'(X), ['$VAR'(X)]) :- !.
get_vars(A, []) :- atom(A), !.
get_vars([A | B], V) :- !,
     get_vars(A, Av),
     get_vars(B, Bv),
     append(Av, Bv, V).
get_vars(F, V) :-
     F =.. [_ | A],
     !,
     get_vars(A, V).
get_vars(_, []).
```

```
%
%  rhs_vars(+Term, -Variables)
%    Variables is a list of variables occurring in Term;
%         variables occurring within a 'where' clause
%         are only listed if they appear on the RHS
%         of a '<=' term
%
rhs_vars(T causing _ where W, V) :-
     !,
     rhs_vars(T, RV),
     where_vars(W, WV),
     append(RV, WV, V).
rhs_vars(T where W, V) :-
     !,
     rhs_vars(T, RV),
     where_vars(W, WV),
     append(RV, WV, V).
rhs_vars(T causing _, V) :-
     !,
     rhs_vars(T, V).
rhs_vars(T, V) :-
     get_vars(T, V).

%
%  where_vars(+Term, -Variables)
%    Variables is a list of variables that occur on the
%    RHS of a '<=' rule in Term
%
where_vars((W1, W2), V) :-
     !,
     where_vars(W1, V1),
     where_vars(W2, V2),
     append(V1, V2, V).
where_vars(_ <= T, V) :-
     !,
     rhs_vars(T, V).
where_vars(_, []).
```

```
%
%  newRule(+LHS, +RHS, +VariablesToCancel, -NewRule)
%    NewRule is the Log(F) rule 'LHS => NewRHS' where
%         NewRHS is formed by adding a cancel goal for
%         each variable occurring in VariablesToCancel
%         to the 'causing' portion of RHS.  If no
%         'causing' portion exists, one is added
%
newRule(LHS, RHS, [], (LHS => RHS)) :- !.
newRule(LHS, T causing C where W, CV,
          (LHS => T causing C2 where W)) :-
     !,
     add_CV(C, CV, C2).
newRule(LHS, T causing C, CV, (LHS => T causing C2)) :-
     !,
     add_CV(C, CV, C2).
newRule(LHS, T where W, CV,
     (LHS => T causing C2 where W)) :-
          !,
          add_CV([], CV, C2).
newRule(LHS, T, CV, (LHS => T causing C2)) :-
     !,
     add_CV([], CV, C2).

%
%  add_CV(+CancelClause, +Variables, -NewCancelClause)
%    NewCancelClause is formed by adding a cancel goal
%         to CancelClause for each variable
%         occurring in Variables
%
add_CV(C, [], C).
add_CV([], [H | T], C2) :-
     !,
     add_CV(cancel(H), T, C2).
add_CV(C, [H | T], C2) :-
     add_CV((C, cancel(H)), T, C2).
```

```
%
% cso(+Rule, -NewRule)
%     NewRule is formed as follows. For each term of the
%     form 'V <= S' occurring in Rule, do the following:
%     if S = #T, replace 'V <= S' by the Prolog goal,
%          remote(T, NC, V).
%     if S = T@H, replace 'V <= S' by the goal
%          remoteSite(H, T, NC, V).
%     if S = lazyRemote(N, M, T), replace 'V <= S' by
%          the goal lazyRemote(N, M, T, NC, V).
%     if S = lazyRemoteSite(H, N, M, T), replace 'V <= S'
%          by lazyRemoteSite(H, N, M, T, NC, V).
%     (In all of the above, NC is the number of processes
%      specified in Rule which consume V.  If V occurs
%      multiple times within a single process, common
%      subexpression evaluation is performed, as
%      described below.)
%     otherwise, S = G. G is a local common subexpression.
%          All occurrences of S are replaced by
%          common(T, G, L), where T is a common variable
%          shared by all NC occurrences of the common
%          subexpression in Rule and L is a list of NC
%          variables.  L is used when cancelling streams
%          to assure that a stream is actually cancelled
%          only after all of its local consumers have
%          requested cancellation.
%
cso(R, (LHS => RHS2))  :-
     numbervars(R, 0, N),
     R = (LHS => RHS), !,
     cso(RHS, N, _, RHS2).
cso(R, R).

cso(RHS where W, N, N4, RHS5)  :-  !,
     cso(RHS, N, N2, RHS2),   % do inner-most where
                              % clauses first
     cso(W, N2, N3, W2),
     compile_each(0, RHS2 where W2, RHS4, CL, N3, N4),
     reformat(RHS4, CL, RHS5).
cso('$VAR'(X), N, N, '$VAR'(X))  :-  !.
cso(A, N, N, A)  :-
     atom(A).
cso([A | B], N, N3, [A2 | B2])  :-
     cso(A, N, N2, A2),
     cso(B, N2, N3, B2).
cso(F, N, N2, F2)  :-
     F =.. [H | A],
     cso(A, N, N2, A2),
     F2 =.. [H | A2].
```

```
%
% compile_each(+NW, +Term, -NewTerm, -Counts, +N, -N2)
%    perform the compilation process for each term in
%         the 'where' portion of Term to form NewTerm
%    NW indicates which term of the where clause to
%         process next.  Note that these terms must
%         be kept as a part of Term as they themselves
%         may be changed by the compilation process.
%    Counts is a list which indicates how many times
%         each of the stream descriptors in the
%         where clause occurred in Term
%    N and N2 are merely used for numbervarsing terms
%
compile_each(NW, T, T3, [C | CL], N, N3) :-
     numbervars(T, N, N2),
     nth_W(NW, T, WC), !,
     compile_one(WC, T, T2, C),
     NW2 is NW + 1,
     compile_each(NW2, T2, T3, CL, N2, N3).
compile_each(_, T, T, [], N, N2) :-
     numbervars(T, N, N2).

compile_one(V <= #_, T, T2, P)  :-
     !,
     compile_subexpression(T, T2, V, V, P).
compile_one(V <= _ @ _, T, T2, P)  :-
     !,
     compile_subexpression(T, T2, V, V, P).
compile_one(V <= lazyRemote(_, _, _), T, T2, P)  :-
     !,
     compile_subexpression(T, T2, V, V, P).
compile_one(V <= lazyRemoteSite(_, _, _, _),
        T, T2, P)  :-
     !,
     compile_subexpression(T, T2, V, V, P).
compile_one(V <= CT, T, T2, local)  :-
     !,
     compile_subexpression(T, T2, V, CT, _).
compile_one(_, T, T, prolog).
```

```
%
% compile_subexpression(
%    +Term,          Term is compiled to
%    -NewTerm, NewTerm
%    +Variable,      local occurrences of Variable are
%                    replaced by
%    +CommonTerm,    structures that assure minimal
%                    reduction of CommonTerm.
%    -Processes      The total number of processes which
%                    consume Variable is returned
%    )
%
compile_subexpression(T, T2, V, CT, P) :-
     compile(T, V, T2, LV, CT, LO, RO),
     localCSO(LO, LV, CT),
     processes(LO, RO, P).

%
%  processes(LO, RO, Processes)
%    the number of processes which consume a stream is
%    - the number of remote occurrences (RO) if
%         there  are no local occurrences (LO)
%    - RO + 1 if there are any local occurrences
%
processes(0, RO, RO)  :- !.
processes(_, RO, P)  :-
     P is RO + 1.

%
%  localCSO(+LocalOccurrences, -OptimizedExpr, +Expr)
%    if there is more than one local occurrence of Expr,
%         is is replaced by a common/3 structure
%         which guarantees minimality
%    otherwise, it is left as is
%
localCSO(0, _, _)  :-  !.
localCSO(1, V, V)  :-  !.
localCSO(1, common(_, V, _), V).
localCSO(N, common(_, V, L), V) :-
     nList(N, L).
%
%  nList(+N, -L)
%    L is a list of N uninstantiated variables
%
nList(0, [])  :-  !.
nList(N, [_ | B])  :-
     Nminus1 is N - 1,
     nList(Nminus1, B).
```

```
% compile(          compile
%    +Term,         Term
%    +Variable,     in which Variable represents
%                        a subexpression
%    -NewTerm,      into NewTerm
%    +LocalVar,     substituted for local occurrences
%                        of Variable
%    +CommonTerm,   substituted for single occurrences
%                   of Variable
%    -LO,       # of times Variable is used by
%                        local process
%    -Processes,    # of remote processes which use
%                        Variable
%     )

% remote cases
compile(V <= RHS, V, V <= RHS, _, _, 0, 0) :- !.
compile(R <= #T, V, R <= #T2, _, CT, 0, P) :- !,
          compile_subexpression(T, T2, V, CT, P).
compile(R <= lazyRemote(N, M, T), V,
     R <= lazyRemote(N, M, T2), _, CT, 0, P) :- !,
          compile_subexpression(T, T2, V, CT, P).
compile(R <= T@H, V, R <= T2@H, _, CT, 0, P) :- !,
          compile_subexpression(T, T2, V, CT, P).
compile(R <= lazyRemoteSite(H, N, M, T), V,
     R <= lazyRemoteSite(H, N, M, T2),
     _, CT, 0, P) :- !,
          compile_subexpression(T, T2, V, CT, P).
compile(#T, V, #T2, _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(lazyRemote(N, M, T), V,
     lazyRemote(N, M, T2), _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(T@H, V, T2@H, _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(lazyRemoteSite(H, N, M, T), V,
     lazyRemoteSite(H, N, M, T2), _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(remote(T, N, S), V,
     remote(T2, N, S), _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(lazyRemote(N, M, T, NC, S), V,
     lazyRemote(N, M, T2, NC, S), _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(remoteSite(H, T, N, S), V,
     remoteSite(H, T2, N, S), _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
compile(lazyRemoteSite(H, N, M, T, NC, S), V,
     lazyRemoteSite(H, N, M, T2, NC, S),
     _, CT, 0, RO)  :- !,
          compile_subexpression(T, T2, V, CT, RO).
```

```
% other cases

compile(V, V, LV, LV, _, 1, 0).
compile(T, _, T, _, _, 0, 0) :-
     atom(T).
compile([A | B], V, [A2 | B2], LV, CT, LO, RO)  :-
     compile(A, V, A2, LV, CT, LO1, RO1),
     compile(B, V, B2, LV, CT, LO2, RO2),
     LO is LO1 + LO2,
     RO is RO1 + RO2.
compile(T, V, T2, LV, CT, LO, RO)  :-
     T =.. [F | A],
     compile(A, V, A2, LV, CT, LO, RO),
     T2 =.. [F | A2].
```

```
%
%  reformat(+Term, +CountList, -NewTerm)
%    NewTerm is formed by replacing '<=' rules in
%    Term by the appropriate Prolog goals
%    CountList indicates the following about each
%    term in the 'where' portion of Term:
%        1. it is a Prolog goal and should be left as is.
%        2. it is a local common subexpression and should
%           be removed. (It has already been replaced
%           "in-line" within Term)
%        3. It is a remote common subexpression and the
%           element of CountList indicates the number of
%           consumers.
%
reformat(RHS, [], RHS)  :- !.
reformat(RHS where Where, CountList, RHS2)  :-
     change_counts(CountList, Where, Where2),
     eliminate_trues(Where2, Where3),
     make_term(RHS, Where3, RHS2).

change_counts([T], W1, W2) :- !,
     change_one(T, W1, W2).
change_counts([H | T], (W1, Ws1), (W2, Ws2)) :-
     change_one(H, W1, W2),
     change_counts(T, Ws1, Ws2).

change_one(prolog, G,  G).
change_one(local, _, true).
change_one(N, V <= #T, remote(T, N, V)).
change_one(N, V <= T @ H, remoteSite(H, T, N, V)).
change_one(NC, V <= lazyRemote(N, M, T),
          lazyRemote(N, M, T, NC, V)).
change_one(NC, V <= lazyRemoteSite(H, N, M, T),
          lazyRemoteSite(H, N, M, T, NC, V)).

make_term(RHS, true, RHS) :- !.
make_term(RHS, W, RHS where W).

%
%  nth_W(+N, +RHS, -WC)
%    WC is the Nth term in the where clause of RHS
%
nth_W(N, _ where W, WC) :-
     nth_of_W(N, W, WC), !.

nth_of_W(0, (H, _), H) :- !.
nth_of_W(0, H, H) :- !.
nth_of_W(N, (_, T), WC) :-
     Nm is N - 1,
     nth_of_W(Nm, T, WC).
```

```
%
%  once(+Goal)
%    allow Goal to succeed at most one time
%
once(G) :- call(G), !.

%
%  filenamePrefix(+File, +ExpectedSuffix, -Prefix)
%    return a string which when extended with
%         ExpectedSuffix forms the string File
%    if no such string exists, return File
%
filenamePrefix(File,ExpectedSuffix,Prefix) :-
        name(File,FileString),
        append(PrefixString,ExpectedSuffix,FileString),
        name(Prefix,PrefixString),
        !.
filenamePrefix(File,_,File).

%
%  extendedFilename(+Prefix, +Extension, -ExtendedName)
%    ExtendedName is the result of appending the strings
%         Prefix and Extension
%
extendedFilename(Prefix,Extension,ExtendedName) :-
        name(Prefix,PrefixString),
        append(PrefixString,Extension,ExtensionString),
        name(ExtendedName,ExtensionString).

eliminate_trues(A,B) :-
     eliminate_trues1(A,Z),
     eliminate_last_true(Z,B).

eliminate_trues1((true,X),X1) :- !,
     eliminate_trues1(X,X1).
eliminate_trues1((X,Y),(X,Y1)) :- !,
     eliminate_trues1(Y,Y1).
eliminate_trues1(X,X).

eliminate_last_true((A,true),A) :- !.
eliminate_last_true((A,B),(A,Z)) :- !,
     eliminate_last_true(B,Z).
eliminate_last_true(A,A).
```

APPENDIX B

CLIENT IMPLEMENTATION

In this appendix, we present the code for each of the concurrent processing options available to the ASPEN programmer.

```
%
% client.logf
%
% Context to allow remote term reduction.
%

:- op(730, xfy, where).
:- op(730, xfy, causing).
:- op(730, xfy, (<=)).
:- op(730, fy, #).
:- op(730, xfy, @).
:- op(725, yfx, in).

:- ['/usr/local/lib/prolog/lib/sets'].
     % for subtract/3 and memberchk/2

:- ['/usr/local/lib/prolog/lib/lists'].
     % for append/3
% all site names are to be considered constructor symbols

simplified(local).
simplified(thetford).
simplified(exeter).
simplified(warwick).
simplified(windsor).
simplified(wingfield).
simplified(rye).
simplified(raglan).
simplified(dover).
simplified(penzance).
simplified(nottingham).
simplified(kingston).
simplified(ipswich).
simplified(beverly).
```

```
%
%  #(+Term)
%
% Reduce Term remotely.
% (single consumer)
%
#Term  =>  Term @  selectSite(Term).

%
% +Term @ +Host
%
% Reduce 'Term' on the specified  host.
% (single consumer)
%
Term @ Host  =>
     if(Host = local,
          Term,
          remote_stream(Channel)
                    where
               (reduce(Host, HostR),
                connectServer(HostR, Channel),
                     writeStream(Channel, single(Term))
               )
     ).

%
%  lazyRemote(+N, +M, +Term)
%
%  constrained reduction of Term
%  (single consumer)
%
lazyRemote(N, M, Term) =>
     lazyRemoteSite(selectSite(Term), N, M, Term).

%
%  lazyRemoteSite(+Host, +N, +M, +Term)
%
%  constrained reduction of Term on the specified host
%  (single consumer)
%
lazyRemoteSite(Host, N, M, Term)  =>
     if(Host == local,
          Term,
          lazy_remote(Socket, N,  M)
               where
               (reduce(Host, HostR),
                connectServer(HostR, Socket),
                writeStream(Socket, lazy(N, M, Term))
               )
     ).
```

```
%
%  remote(+Term, +N, -Stream)
%
%  Stream is the stream descriptor for the
%  reduction of Term with N consumers
%
remote(Term, N, Stream) :-
     reduce(selectSite(Term), Host),
     remoteSite(Host, Term, N, Stream).


%
%  remoteSite(+Host, +Term, +N, -S)
%
%  S is the stream descriptor for the reduction (on
%  the specified host) of Term with N consumers
%
remoteSite(Host, Term, _, Term) :-
     reduce(Host, local), !.
remoteSite(Host, Term, N, StreamDescriptor) :-
     reduce(Host, HostR),
     connectServer(HostR, Socket),
     writeStream(Socket, multiple(Term, N)),
     readStream(Socket, StreamDescriptor),
     closeStream(Socket).


%
%  lazyRemote(+N, +M, +Term, +NC, -Stream)
%
%  Stream is the stream descriptor for the
%  constrained reduction of Term with NC
%  consumers
%
lazyRemote(N, M, Term, NC, Stream)  :-
     reduce(selectSite(Term), Host),
     lazyRemoteSite(Host, N, M, Term, NC, Stream).


%
%  lazyRemoteSite(+Host, +N, +M, +Term, +NC, -Stream)
%
%  Stream is the stream descriptor for the constrained
%  reduction (on the specified host) of Term with NC
%  consumers
%
lazyRemoteSite(Host, _, _, Term, _, Term) :-
     reduce(Host, local), !.
lazyRemoteSite(Host, N, M, Term, NC, StreamDescriptor) :-
     reduce(Host, HostR),
     connectServer(HostR, Socket),
     writeStream(Socket, lazy(N, M, Term, NC)),
     readStream(Socket, StreamDescriptor),
     closeStream(Socket).
```

```
%
% remote_stream(+Socket)
%
% Reduces to an element off of a stream and the
% continuation of the stream
%
reduce(remote_stream(Socket), Stream) :-
     readStream(Socket, Term),
     !,
     remoteStream(Socket, Term, Stream).

%
% remoteStream(+Socket, +Term, -Stream)
%
%  if Term, which has been read off of Socket indicates
%  an error condition or end_of_stream, the appropriate
%  action is taken.  Otherwise, Term is returned.
%
remoteStream(Socket, '$error', _) :-
     !,
     closeStream(Socket),
     fail.
remoteStream(Socket, '$end_of_stream', []) :-
     !,
     closeStream(Socket).
remoteStream(Socket, '$end_of_substream', []) :- !.
remoteStream(Socket, Term, [Term | remote_stream(Socket)]).

%
% constrained reduction
%
reduce(lazy_remote(Socket, N, M), Stream) :-
     readStream(Socket, Term),
     !,
     lazyRemote(Socket, Term, N, M, Stream).

lazyRemote(Socket, '$end_of_stream', _, _, []) :-
     !,
     closeStream(Socket).
lazyRemote(Socket, Term, 1, M,
          [Term|lazy_remote(Socket, M, M)]) :-
     !,
     writeStream(Socket, resume).
lazyRemote(Socket, Term, N, M,
          [Term|lazy_remote(Socket, Nn, M)]) :-
     Nn is N - 1.
```

```
%
% dedicated servers
%
dedicated_service(Socket, T) =>
     dedicated_host(selectSite(T), Socket, T).

reduce(dedicated_host(Host, M1, Term in M2), Stream) :-
     reduce(Host, local),
     var(M1),
     !,
     active_module(Mold),
     module(M2),
     M1 = M2,
     reduce(Term, Stream),
     module(Mold).
reduce(dedicated_host(Host, M, Term in M), Stream) :-
     reduce(Host, local),
     !,
     active_module(Mold),
     module(M),
     reduce(Term, Stream),
     module(Mold).
reduce(dedicated_host(Host, Socket, Term), Stream) :-
     isSocket(Socket),
     !,
     writeStream(Socket, Term),
     reduce(remote_stream(Socket), Stream).
reduce(dedicated_host(Host, Socket, Term), Stream) :-
     reduce(Host, HostR),
     connectServer(HostR, Socket),
     writeStream(Socket, dedicated(Term)),
     reduce(remote_stream(Socket), Stream).


%
% common subexpressions
%

reduce(common(T, S, L), R) :-
     nonvar(T),
     !,
     R = T.
reduce(common(R, S, L), R) :-
     reduce(S, U),
     common(U, L, R).

common([A|B], L, [A|common(_, B, L)])  :-  !.
common(T, L, T).
```

```
%
% +Term where +Condition
%
%  reduce Term after calling Condition
%
reduce(where(Term, Condition), NewTerm) :-
     call(Condition),
     reduce(Term, NewTerm).

%
%  +Term causing +Action
%
%  perform Action after reducing Term
%
reduce(causing(Term, Action), NewTerm) :-
     reduce(Term, NewTerm),
     call(Action).


%
% first(+Stream) and second(+Stream)
%  filter the appropriate type out of Stream
%
first([]) =>  [].
first([E|R]) =>
     if(E = o1(T), [T|first(R)], first(R)).

second([]) => [].
second([E|R]) =>
     if(E = o2(T), [T|second(R)], second(R)).


% streamDescriptor(+Host, +Port)
%
% Reduces to a open channel to another process at 'Host'
% and 'Port'.
%
reduce(streamDescriptor(Host, Port), Stream) :-
     connectStream(Port, Host, Channel),
     reduce(remote_stream(Channel), Stream).


isSocket(X) :- integer(X).
```

```
%
%  select(+List)
%  List contains 0 or more terms to be reduced locally
%  and 0 or more terms to be reduced remotely
%    - all remote reductions are started immediately
%    - upon each call, if there is data available on a
%      stream from a remote process, that term is
%      returned.  Otherwise, one of the local
%      reductions is performed
%
reduce(select(L), R) :-
     split(L, Local, Remote),
     reduce(select3(Remote, Local, non_blocking), R).

reduce(select3(Remote, Local, M), Result) :-
     selectStreams(Remote, Ready, Closed, M),
     subtract(Remote, Closed, New),
     localOrRemote(Ready, New, Local, Result).

%
%  If a remote term is available, return it
%  Otherwise, perform a local reduction
%
localOrRemote([], [], [], []).
localOrRemote([], [R|Rs], [], Result) :-
     reduce(select3([R|Rs], [], blocking), Result).
localOrRemote([], Streams, [A|B], Result) :-
     reduce(A, Ar),
     localResult(Ar, B, Streams, Result).
localOrRemote([R|_], Streams, Local, Result) :-
     readStream(R, Term),
     processTerm(Term, R, Streams, Local, Result).

%
%  process a term read off of a stream
%
processTerm('$error', Socket, Streams, Local, R) :-
     !,
     closeStream(Socket),
     subtract(Streams, [Socket], Streams2),
     reduce(select3(Streams2, Local, non_blocking), R).
processTerm('$end_of_stream', Socket, Streams, Local, R) :-
     !,
     closeStream(Socket),
     subtract(Streams, [Socket], Streams2),
     reduce(select3(Streams2, Local, non_blocking), R).
processTerm(Term, Socket, Streams, Local,
     [Term|select3(Streams, Local, non_blocking)]).
```

```
%
%  process the result of a local reduction
%
localResult([], Local, Remote, Result) :-
     reduce(select3(Remote, Local, non_blocking), Result).
localResult([A|B], Local, Remote,
     [A|select3(Remote, [B|Local], non_blocking)]).

%
%  split(+Terms, -Local, -Remote)
%
%  Remote is a list of stream descriptors for processes
%    allocated to reduce annotated terms in Terms
%  Local is a list of unannotated terms in Terms
%
split([], [], []).
split([#T|Ts], L, [Channel|R]) :-
     !,
     reduce(selectSite(T), H),
     connectStream(1550, H, Channel),
     writeStream(Channel, single(T)),
     split(Ts, L, R).
split([T @ H|Ts], L, [Channel|R]) :-
     !,
     connectStream(1550, H, Channel),
        writeStream(Channel, single(T)),
     split(Ts, L, R).
split([T|Ts], [T|L], R) :-
     split(Ts, L, R).

%
%  selectStreams(+Streams, -Ready, -Closed, +Mode)
%
%  call builtin selectStreams/8 to see if data is
%  available on any of the streams in Streams.  Ready
%  is the subset of Streams upon which data is available.
%  Closed is the subset of Streams upon which exceptions
%  occurred.  Mode indicates whether the call to
%  selectStreams should block or return immediately if
%  no data is available
%
selectStreams(Streams, Ready, Closed, Mode) :-
     selectStreams(_, Streams, [], [], Ready, _,
          Closed, Mode).
```

```
%
%  cancel(+Term)
%
%  close all streams that occur in Term
%    - streams that occur in either of the two forms
%      remote_stream(S) or remote_stream(S, _, _)
%      may be closed immediately
%    - streams that occur in the form
%      streamDescriptor(Host, Port) must first be
%      connected, then closed.  This assures that a
%      process will not wait forever for a consumer
%      that will never arrive
%    - streams that occur within common(T, S, L)
%      structures must be treated very carefully; first,
%      we instantiate one of the variables in L to
%      closed (this will be seen by all instances of
%      the common structure since the variables are
%      shared.  If all of the variables in L have been
%      instantiated, we may close S.
%
cancel(X)  :-
     var(X), !.
cancel(X)  :-
     atom(X), !.
cancel(X)  :-
     current_module(X), !, drop_module(X).
cancel(remote_stream(S))  :- !,
     closeStream(S).
cancel(remote_stream(S, _, _))  :-  !,
     closeStream(S).
cancel(streamDescriptor(Host, Port))  :-  !,
     connectStream(Port, Host, Stream),
     closeStream(Stream).
cancel(common(_, S, L))  :-  !,
     mark_one(L),
     cancel_if_last(L, S).
cancel([ ]).
cancel([A | B])  :-
     cancel(A),
     cancel(B).
cancel(Term)  :-
     Term  =..  [_ | Args],
     cancel(Args).

mark_one([ ]).
mark_one([A | _])  :-
     var(A), !,
     A = cancelled.
mark_one([_ | B])  :-
     mark_one(B).
```

```
cancel_if_last(L, S)  :-
     all_marked(L), !,
     cancel(S).
cancel_if_last(_, _).

all_marked([A | B])  :-
     nonvar(A),
     all_marked(B).
all_marked([ ]).


%
% selectSite(+Term)
%
%  Reduces to a site upon which to reduce the given term.
%  The site is selected in a round-robin fashion from a
%  list of available sites

reduce(selectSite(_), S) :-
     siteList([S|Ss]),
     retract(siteList/1),
     append(Ss, [S], NSs),
     assert(siteList(NSs)).

siteList([thetford, exeter, warwick, windsor, wingfield,
     rye, raglan, dover, bath, york, penzance,
     nottingham, kingston, ipswich, beverly]).

connectServer(Host, Socket) :-
     connectStream(1550, Host, Socket).
          % all servers are located at port 1550.
```

APPENDIX C

WORKER IMPLEMENTATION

In this appendix, we present the full Prolog specification of the worker process. It is interfaced by clients via the messages described in Section 4.1.2.1. Recall that those messages were **`single(Term)`**, **`multiple(Term, N)`**, **`lazy(N, M, Term)`**, **`lazy(N, M, Term, NC)`**, and **`dedicated(Term)`**.

```
%
% worker.pl
%
% Assumes the existence of the following structure
% in the clause database as established by the server :
%
% '$workerData'(RequestPortSock, IdleSemdId,
%         LockSemId, WorkerNum)
%

%
% '$acceptRequests'
%
% This procedure waits for a request of service to
% be handed to it from the server.
%

     '$workerData'(RequestPortSock, IdleSemId, LockSemId,
                   WorkersFreeSemId, WorkerNum),
     signalSemFree(IdleSemId, WorkerNum),
     signalSemFree(WorkersFreeSemId, 0),
     waitOnSemNotFree(IdleSemId, WorkerNum),
     acceptStream(RequestPortSock, CS),
     signalSemFree(LockSemId, 0),
     readStream(CS, Request),
     handleRequest(Request, CS),
     '$acceptRequests'.
```

```
%
% handleRequest(Request, ClientStream)
%
% handle Request which has arrived on ClientStream
%

%
% Request = single(Term)
%    - request is for the eager reduction of Term
%    - requesting process is the the only consumer
%
handleRequest(single(Term), CS) :-
     makeStream(Term, CS).

%
% Request = lazy(N, M, Term)
%    - request is for the constrained reduction of Term
%    - requesting process is the the only consumer
%
handleRequest(lazy(N, M, Term), S) :-
     lazyStream(N, M, Term, S).

%
% Request =  multiple(Term, N)
%    - request is for the eager reduction of Term
%    - the results are to be sent to N consumers
%
handleRequest(multiple(Term, N), S) :-
     initialize(Term, N, S, Buffer, ServiceID),
     eagerLoop(Buffer, ServiceID).

%
% Request = lazy(N, M, Term, NC)
%    - request is for the constrained reduction of Term
%    - the results are to be sent to NC consumers
%
handleRequest(lazy(N, M, Term, NC), S) :-
     initialize(Term, NC, S, Buffer, ServiceID),
     molazyStream(N, M, Buffer, ServiceID).

%
% Request = dedicated(Term)
%    - request is for the allocation of a dedicated worker
%    - Term is the first term to be reduced by the
%      allocated worker
%
handleRequest(dedicated(Term), S) :-
     dedicatedService(Term, S).
```

```
%
% makeStream(Term, Stream)
%    - Term is reduced and the results are sent on Stream
%      to a single consumer
%

makeStream('$done', Stream) :- !,
     closeStream(Stream).
makeStream(Term, Stream) :-
     reduce(Term, ReducedTerm), !,
     action(ReducedTerm, NewTerm, Stream),
     makeStream(NewTerm, Stream).
makeStream(Term, Stream) :-
          % reduction failed -> error
     writeStream(Stream, '$error'),  !,
     closeStream(Stream).
makeStream(Term, Stream)  :-
          % write failed -> client closed?
     closeStream(Stream).

action([], '$done', Stream) :-
     writeStream(Stream, '$end_of_stream'), !.
action([H|T], T, Stream) :-
     writeStream(Stream, H), !.
action(_, '$done', _).   % write failed -> client closed?
```

```
%
% lazyStream(N, M, Term, Stream)
%         N is the number of solutions that should be
%              sent initially
%         M is the number of solutions that should be
%              sent in response to each 'resume' message
%         Term is the term to be reduced to produce the
%              next binding
%         Stream is the stream on which output
%              is to be sent
%
lazyStream(_, _, [], Stream) :-  !,
     writeStream(Stream, '$end_of_stream'),
     closeStream(Stream).
lazyStream(N, M, Term, Stream) :-
     mSend(N, Term, Term2, Stream),
     readStream(Stream, resume),  !,
     lazyStream(M, M, Term2, Stream).
lazyStream(_, _, _, Stream) :-
     closeStream(Stream).

%
% mSend(M, Tin, Tout, Stream)
%    M is the number of bindings to produce
%    Tin is the initial term before any reductions
%    Tout is the final term after M reductions
%    Stream is the output stream
%
mSend(_, [], [], Stream) :-
     writeStream(Stream, '$end_of_stream'),
     !,
     closeStream(Stream).
mSend(_, [], [], Stream) :-
     !,
     closeStream(Stream).
mSend(0, Term, Term, _) :- !.
mSend(M, Term, Term2, Stream) :-
     reduce(Term, ReducedTerm),  !,
     recurse_constrained(ReducedTerm, Term2, Stream, M).
mSend(_, _, _, Stream)  :-
     writeStream(Stream, '$error'),  !,
     closeStream(Stream).
mSend(_, _, _, Stream)  :-
     closeStream(Stream).

recurse_constrained([], [], Stream, _) :- !,
     closeStream(Stream).
recurse_constrained([H|T], T2, Stream, M) :-
     writeStream(Stream, H),  !,
     Mminus1 is M - 1,
     mSend(Mminus1, T, T2, Stream).
```

```
% initialize(
%    +Term,         % term to be reduced
%    +N,            % number of consumers to expect
%    +ClientStream, % stream to client who requested service
%    -Buffer,  % structure that is to contain all necessary info
%              %  buffer(
%              %    Term,     % remainder of term to be reduced
%              %    Tail,     % pointer to end of buffer
%              %    List,     % buffers result until all
%              %         %    consumers have arrived
%              %    CL,  % list of consumers and their
%              %         %    associated pointers
%              %         %    into the buffer
%              %    NC   % number of consumers who have
%              %         %    not yet arrived
%              %    )
%    - ServiceID    % address at which worker will receive
%         %    connection requests from consumers
%    )
%
%
initialize(Term, N, ClientStream,
          buffer(Term, T, T, [ ], N), ServiceID) :-
     establishService(P, ServiceID),
     getHostName(H),
     writeStream(ClientStream, streamDescriptor(H, P)),
     closeStream(ClientStream).

%
%  eagerLoop(+Buffer, +ServiceID)
%
%  eagerly produce output for each of the consumers in
%  Buffer by reducing the term in Buffer.  Reduction
%  stops only when the term has been completely reduced
%  or all consumers have cancelled.  If the term is
%  completely reduced and there are still consumers
%  outstanding, results are buffered for them until
%  they have accepted all results or cancelled.
%
eagerLoop(buffer([], [], [], [], 0), ServiceID) :- !,
     shutdownService(ServiceID).
eagerLoop(Buffer, ServiceID) :-
     newConsumers(ServiceID, Buffer, Buffer2),
     reduceTerm(Buffer2, Buffer3),
     sendOutput(Buffer3, Buffer4), !,
     eagerLoop(Buffer4, ServiceID).
```

```
%
%  buffer manipulation predicates
%

%
%  newConsumers(+ServiceID, +Buffer1, -Buffer2)
%
%  check to see if there are any new consumers waiting
%  on ServiceID
%    - if so, add them to the consumer list in Buffer
%    - note that a non-blocking select is used; if no
%      new consumers have arrived since last we checked,
%      we do not wait for one to arrive
%    - streams established with new consumers are set to
%      allow non-blocking I/O so that writes to those
%      streams will return immediately if the IPC buffer
%      is full
%
newConsumers(SID, buffer(Cont, T, L, C, 0),
     buffer(Cont, T, L, C, 0)) :-  !.
newConsumers(SID, B, B) :-
     selectStreams(_, [SID], [], [],
               A, _, _, non_blocking),
     A == [],  !.
newConsumers(SID, B1, B3) :-
     acceptStream(SID, S),
     setNoBlock(S),
     addConsumer(B1, S, B2),
     newConsumers(SID, B2, B3).

%
%  addConsumer(+Buffer1, +Stream, -Buffer2)
%
%  add a newly arrived consumer, with whom a connection,
%  Stream, has been established, to Buffer.  Note that
%  when the last consumer arrives (N becomes 0), the
%  List argument of Buffer1 is set to [] so that
%  garbage collection will begin for elements in the
%  buffer that have already been sent to all consumers
%
addConsumer(buffer(Cont, T, L, C, 1), S,
         buffer(Cont, T, [ ], [c(L, S)|C], 0)) :-
     !.
addConsumer(buffer(Cont, T, L, C, N), S,
         buffer(Cont, T, L, [c(L, S)|C], M)) :-
     M is N - 1.
```

```
%  reduceTerm(+Buffer1, -Buffer2)
%
%  the term in Buffer1 is reduced
%  Buffer2 reflects the result of this reduction
%    the continuation and tail are updated appropriately
%
reduceTerm(buffer(Cont, T, L, C, N), NewB) :-
     reduce(Cont, ReducedCont),
     matchTerm(ReducedCont, buffer(_, T, L, C, N), NewB).

matchTerm([],
     buffer(_, [ ], L, C, N),
     buffer([ ], [ ], L, C, N)).
matchTerm([A | NewCont],
     buffer(Cont, [A | NewT], L, C, N),
     buffer(NewCont, NewT, L, C, N)).

%
%  sendOutput(+Buffer1, -Buffer2)
%
%  an attempt is made to send data to each consumer in
%  Buffer1 for whom data is available.  If a term is
%  successfully sent, the buffer is updated accordingly.
%  If the write would block because the IPC buffer for
%  that consumer is full, the buffer remains unchanged
%  and another attempt to send the data will be made
%  later. If a write to a particular consumer fails,
%  that consumer will be removed from the buffer structure.
%
sendOutput(buffer(Cont, Tail, List, CL, N),
          buffer(Cont, Tail, List, CLp, N)) :-
     sendOnAll(CL, CLp).

sendOnAll([ ], [ ]).
sendOnAll([c(L, S) | CS], [c(L, S) | CSp]) :-
     var(L),  !,          % no data available for
                          % this consumer
     sendOnAll(CS, CSp).
sendOnAll([c([H | Lp], S) | CS], CSp) :-
     writeStreamNB(S, H, RV), !,
     continueSend(RV, S, H, Lp, CS,  CSp).
sendOnAll([c([ ], S) | CS], CSp) :-
     writeStreamNB(S, '$end_of_stream', RV), !,
                          % this consumer has seen
                          % the entire stream
     continueEOS(RV, S, CS, CSp).
sendOnAll([c(_, S) | CS], CSp) :-
     closeStream(S),      % a writeStream failed
                          % close the stream and
                          % remove the consumer
     sendOnAll(CS, CSp). % from the buffer structure
```

```
%
%  continueSend(+ReturnValue, +Stream, +Term, +Rest,
%    +OtherConsumers, -NewConsumers)
%
%  if the write of Term to Stream would have blocked
%  (ReturnValue = would_block), put Term back into the
%  buffer for Stream and continue with the other
%  consumers. Otherwise (ReturnValue = success), attempt
%  to send more data elements from Rest on Stream before
%  proceeding to the other consumers.
%
continueSend(would_block, S, H, Lp, CS,
          [c([H | Lp], S) | CSp])  :-
     sendOnAll(CS, CSp).
continueSend(success, S, _, Lp, CS, CSp)  :-
     sendOnAll([c(Lp, S) | CS], CSp).

%
%  continueEOS(+ReturnValue, +Stream,
%    +OtherConsumers, -NewConsumers)
%
%  if the write of '$end_of_stream' to Stream would have
%  blocked (ReturnValue = would_block), put [] back into
%  the buffer for Stream and continue with the other
%  consumers.
%  Otherwise (ReturnValue = success), remove Stream
%  from the list of consumers
%
continueEOS(would_block, S, CS, [c([ ], S) | CSp])  :-
     sendOnAll(CS, CSp).
continueEOS(success, _, CS, CSp)  :-
     sendOnAll(CS, CSp).


%
%  end of buffer manipulation predicates
%
```

```
%
%  moLazyStream(+N, +M, +Buffer, +ServiceID)
%
%  lazyStream with multiple outputs
%
%    N is the number of solutions that should be
%              sent initially
%    M is the number of solutions that should be
%              sent in response to each 'resume' message
%    Buffer contains the information necessary
%         for producing, buffering, and sending
%         results.
%    ServiceID is the address at which new consumers
%         will be accepted
%
moLazyStream(_, _, [ ], _)  :-  !.
moLazyStream(N, M, Buffer, ServiceID)  :-
     lazyLoop(Buffer, Buffer2, N, ServiceID),
     waitOnAll(Buffer2, Buffer3, ServiceID),
     moLazyStream(M, M, Buffer3, ServiceID).

lazyLoop(buffer([], [], [], [], 0), [], _, ServiceID)  :-
     !,
     shutdownService(ServiceID).
lazyLoop(Buffer, Buffer, 0, ServiceID)  :-  !.
lazyLoop(Buffer, Buffer5, N, ServiceID)  :-
     newConsumers(ServiceID, Buffer, Buffer2),
     reduceTerm(Buffer2, Buffer3),
     Nm is N - 1,
     sendOutput(Buffer3, Buffer4),
     lazyLoop(Buffer4, Buffer5, ServiceID).


%
%  waitOnAll(+ServiceID, +Buffer1, -Buffer2)
%
%  Wait for a 'resume' message to arrive from all consumers
%  in Buffer1.  If not all consumers have arrived yet, we
%  must now wait until they do before proceeding.
%
waitOnAll(ServiceID, buffer(Cont, T, L, C, 0),
          buffer(Cont, T, L, Cp, 0)) :-  !,
     waitOnEach(C, Cp).
waitOnAll(ServiceID, Buffer, Buffer2) :-
     newConsumers(ServiceID, Buffer,
          buffer(Cont, T, L, C, N)),
     waitOnEach(C, Cp),
     waitOnAll(ServiceID, buffer(Cont, T, L, Cp, N),
          Buffer2).
```

```
%
%  waitOnEach(+Consumers1, -Consumers2)
%
%  for each consumer in Consumers1, wait for a 'resume'
%  message to arrive.  If the stream associated with a
%  particular consumer is closed or another type of message
%  arrives before the 'resume' message, the stream to that
%  consumer is closed and the consumers is removed from
%  Consumers1
%
waitOnEach([ ], [ ]).
waitOnEach([c(T, S) | CS], [c(T, S) | CSp]) :-
     readStream(S, resume),  !,
     waitOnEach(CS, CSp).
waitOnEach([c(_, S) | CS] , CSp) :-
     closeStream(S),
     waitOnEach(CS, CSp).
```

```
%
%  dedicatedService(+Term, +Socket)
%
%  A worker is dedicated to a given client to
%  reduce a succession of terms until the client
%  sends a cancel message instead of a term.
%  The worker initially loads the requested
%  context for the reduction and does not
%  unload it until the client relinquishes the
%  dedicated service
%
dedicatedService(cancel, Socket) :-
     !,
     writeStream(Socket, '$end_of_stream'),
     closeStream(Socket).
dedicatedService(Term in Module, Socket) :-
     !,
     active_module(M1),
     module(Module),
     dedicatedService(Term, Socket),
     drop_module(Module),
     module(M1).

dedicatedService(Term, Socket) :-
     reduce(Term, ReducedTerm),
     dedicatedRecurse(ReducedTerm, Socket).

dedicatedRecurse([], Socket) :-
     writeStream(Socket, '$end_of_substream'),
     readStream(Socket, NextTerm),
                         % if the term has reduced
                         % to [], inform the client
                         % and await the next term
                         % to be reduced
     nextTerm(NextTerm, Socket).
dedicatedRecurse([H|T], Socket) :-
     writeStream(Socket, H),
     dedicatedService(T, Socket).

nextTerm(Term in Context, Socket) :-
     !,                       % avoid reloading Context
     dedicatedService(Term, Socket).
nextTerm(Term, Socket).
```

```
%
%  contexts - right now the context mechanism is
%  very primitive.  A modules implementation will
%  certainly help.
%

T in C => S
     causing ( drop_module(C), module(M) )
     where ( active_module(M), module(C) ).

module([]).
module([H | T])  :-
     consult(H),
     module(T).

active_module([]).

drop_module(_).
```

REFERENCES

[1] Abelson, H. and G. Sussman, *The Structure and Analysis of Computer Programs,* MIT Press, Boston, MA (1985).

[2] Barth, Paul, Scott Guthery, and David Barstow, ''The Stream Machine: A Data Flow Architecture for Real-Time Applications,'' pp. 103- 110 in *Proceedings 8th International Conference on Software Engineering*, London, England (August 1985).

[3] Chau, L., ''Functional Grammars and Stream Pattern Matching,'' Draft, UCLA Computer Science Dept. (March 1988).

[4] Gregory, S., *Parallel Logic Programming in PARLOG: The Language and its Implementation,* Addison-Wesley, Reading, MA (1987).

[5] Henderson, Peter, *Functional Programming: Application and Implementation,* Prentice-Hall, Englewood Cliffs, New Jersey (1980).

[6] Kahn, Gilles and David B. MacQueen, ''Coroutines and Networks of Parallel Processes,'' *Proceedings of the IFIP Congress 77*, pp.993-998 (1977).

[7] Levy, Jacob and Ehud Shapiro, ''CFL - A Concurrent Functional Language Embedded in a Concurrent Logic Programming Environment,'' CS86-28, The Weizmann Institute of Science, Rehovot, Israel (December 1986).

[8] Li, P-Y.P. and A.J. Martin, ''The Sync Model: A Parallel Execution Method for Logic Programming,'' *Proc. Symp. on Logic Programming*, pp.223-234, IEEE Computer Society (1986).

[9] Miyazaki, T., A. Takeuchi, and T. Chikayama, ''A Sequential Implementation of Concurrent Prolog Based on the Shallow Binding Scheme,'' pp. 110-118 in *Proceedings IEEE Symposium on Logic Programming*, Boston, MA (1985).

[10] Muntz, R.R. and D.S. Parker, ''Tangram: Project Overview,'' Technical Report CSD-880032, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (April 1988).

[11] Narain, S., "Log(F): A New Scheme for Integrating Rewrite Rules, Logic Programming and Lazy Evaluation," CSD-870027, UCLA Computer Science Dept., Los Angeles, CA (1987).

[12] 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).

[13] Page, T.W., "Prolog Basis for a Data-Intensive Modeling Environment," Dissertation Prospectus, UCLA Computer Science Department, Los Angeles, CA 90024-1596 (March 1988).

[14] Parker, D.S., R.R. Muntz, and L. Chau, "The Tangram Stream Query Processing System," Technical Report CSD-880025, UCLA Computer Science Dept., Los Angeles, CA 90024-1596 (March 1988).

[15] Shapiro, E.Y., *Concurrent Prolog: Collected Papers,* MIT Press, Cambridge, MA (1987).

[16] Sterling, L. and E. Shapiro, *The Art of Prolog,* MIT Press, Cambridge, MA (1986).