Efficient compilation of array expressions

Paul Cockshott
wpc@dcs.gla.ac.uk

Abstract

This article investigates some of the problems involved in generating efficient code for array expressions. It considers issues of cache pollution, effective register use, and vector instructions. It considers the impacts that such compilation efficiency issues can have of the definition of imperative array languages.

Introduction

Let us consider compiled programming languages that allow generalised array expressions, for example F[5], ZPL[6] and Vector Pascal[4]. These all allow expressions of the form x⊕y where ⊕ is drawn from some set of arithmetic operators and x, y are either array variables or other array expressions. This class or languages derives its inspiration from APL which is generally implemented as an interpretive language and which has inspired a family of other interpretive languages J, K, A+.

In order to bring out some of the issues involved in compiling such expressions let us first consider how an interpretive language might handle them. Initially consider the case where ⊕ is a primitive operator like addition or multiplication. If the interpreter is written in C then the array expression can be evaluated by a generalised function of the form:

```
arrayt eval(arrayt x, arrayt y, char omega)
```

where `arrayt` is a type representing a pointer to an array descriptor on the heap. When the function `eval` returns it will have allocated a new array on the heap into which it will have written the result of applying the operation omega to corresponding elements of x and y. If we abstract from the problem of handling arrays of different rank, and consider initially the case where x and y are vectors then this can be done with the following algorithm:

Algorithm 1: Interpretive array expression evaluation

Find L the minimum of the lengths of x and y.
Allocate a new array R of length L.
For i = 0 to L-1 set $R_i = primeval(x_i, y_i, \oplus)$
Return R

Here primeval is a function that evaluates the scalar expression a⊕b, for scalars a,b. We chose an interpretation of x⊕y that returns an array whose length is the minimum of the lengths of x,y for simplicity of explanation.

This is clearly a very naive interpretive algorithm since it costs a function call for each primitive evaluation. Efficiency can be greatly improved by using a case statement to branch on ⊕ prior to for loops:

```
switch(omega){
case ('+'):for(i=0;i<L;i++)R[i]=x[i]+y[i]; break;
case ('-'):for(i=0;i<L;i++)R[i]=x[i]-y[i]; break;
---etc...
```

In this case, the efficiency of the interpretive code can be quite high, indeed it appears to tend to the limiting efficiency of compiled code as L→∞. This sort of optimisation explains why the performance of good interpretive array languages can be comparable with compiled code.

However the final run time performance of a program though crucially dependent on the number of arithmetic instructions in inner loops, the optimisation addressed above, also depends upon other factors. Among these are the efficient use of the storage hierarchy and the ability to exploit vector instructions.

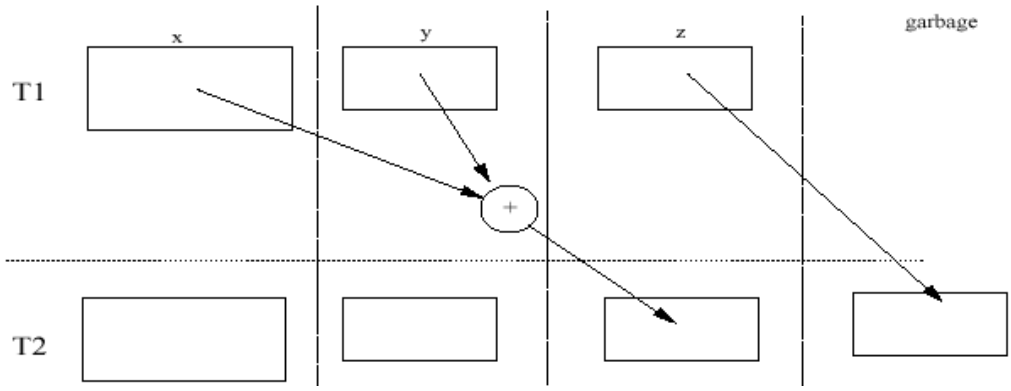

Figure 1: Data flow for `z←x+y`.

Suppose we evaluate the assignment statement `z←x+y`, then the data flow is as shown in Fig 1. At time T1 prior to the execution of the statement we have 3 buffers in memory corresponding to the arrays referenced by the three variables. We disregard the case where z has no initial value. It can be seen that the operation results in the discarding of the original buffer for z which becomes garbage. In a long running algorithm this will have to be recovered. This is storage overhead becomes more significant when we consider more complex array expressions like

```
z ← a*(b-c)
```

In this case a temporary array will be created for the result of `b-c` prior to the array multiply. The first order cost of recovering the space will depend on the sort of storage management algorithm used, and in particular on whether the algorithm always clears any buffer that it allocates. If it is not cleared, then the allocation and recovery will be independent of the buffer size. If the buffer is cleared, then this imposes an overhead of at least one memory store for each pair of primitive operands evaluated in an array expression. By itself, this is a relatively small overhead which can be avoided if there is a non-initialising call to the buffer allocation API. But it becomes more significant once we consider the interaction of buffer allocation with the memory hierarchy.

The levels to consider are registers, cache and main memory and in very large applications virtual memory. The interpretive implementation makes no attempt to exploit registers and makes very inefficient use of the cache. Since the result of each primitive binary operation is written to memory, and since the memory block is always newly allocated, it will tend not to be present in cache. When it is written to for the first time, and in this approach buffers tend to be written to only once, the result will be stored in cache. By the time the multiplication in `a*(b-c)` is done, the result of the subtraction is in cache, so the multiplication proceeds relatively fast. But a cost of storing the result of subtraction in cache is that an equivalent number of bytes of other data must be purged from the cache. If the arrays are sufficiently large relative to cache sizes, then the result of the subtraction could flush the array a out of the cache. Given that access to main memory can be 20 times slower than cache access such cache flushing could seriously affect performance.

A naive array expression compiler would generate code whose dynamic execution would model the dynamic execution of an interpreter. We will model the compilation process by producing equivalent C code. Clearly one route to compilation is to translate the array code into C and leave the machine dependent part of the compilation to the code generator. We will see later that this is not always optimal. The statement

```
z ← a*(b-c)
```

would map to the code shown in Algorithm 2.

Algorithm 2: Naive compiled algorithm

```
t1=memalloc(b_min_c);
for(i=0;i<b_min_c;i++)t1[i]=b[i]-c[i];
t2=memalloc(t1_min_a);
for(i=0;i<t1_min_a;i++)t2[i]=a[i]*t1[i];
dispose(z);
z=t2;
```

For simplicity we assume that the sizes of the result arrays b_min_c etc, have been precomputed either at compile time or in an earlier code fragment. This C code, whilst it has removed the interpretive overheads still retains the same cache penalties as the interpretive code and memory allocation penalties as the original interpretive version.

An alternative approach would be to decompose the array operations so that the whole expression was evaluated as a single loop equivalent to the C code shown in Algorithm 3. This approach, termed drag-through, was described by Budd[3] and by Abrams[1] and has a number of advantages.

Algorithm 3: Optimal version

```
for(i=0;i<zlen;i++)
  z[i]=a[i]*(b[i]-c[i]);
```

In this example we have actually carried out two transformations, we have removed the redundant creation of temporary vectors, and also re-organised the code so that the final result is written into the original buffer z, rather than a newly created buffer z. This optimisation is possible so long as z already references a buffer of the appropriate size, something that a compiler can usually determine.

The expression a[i]*(b[i]-c[i]) is a scalar one, and as such can be effectively computed within registers. This gives us two advantages:

1. Since the temporary result of b[i]-c[i] is in a register, it can be accessed without any store accesses when needed for the subsequent multiply.

2. Since no cache writes occur for the vector result of b-c, the elements of the array a are more likely to be in cache when needed.

vector length	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
Naive μs	0.053	0.047	0.056	0.082	0.088	0.105
optimal μs	0.036	0.032	0.037	0.048	0.057	0.071
naive/optimal	1.48	1.48	1.49	1.70	1.53	1.48

Table 1. Crusoe performance on x:=a*(b-c) implemented in registers versus naively (see table 1). Code generated using 16 bit numbers with MMX vectorisation enabled. Total number of integer operations were held constant as the vector length varies by altering the number of times the operation is repeated. Times are in microseconds to perform the arithmetic to produce each 16 bit result averaged over a large number of runs on a 750MHz Crusoe processor with a 64KB primary cache. Compiled using the Vector Pascal compiler. The naive version was emulated by splitting the statement into two parts thus: f:=b-c; x:=a*f;

vector length	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
naive μs	0.016	0.02	0.021	0.029	0.029	0.03
optimal μs	0.009	0.012	0.0019	0.013	0.014	0.015
naive/optimal	1.66	1.61	1.11	2.2	2.13	2.03

Table 2. P4 performance on x:=a*(b-c) implemented in registers versus naively (see table 2). Vectorisation was enabled. Times are in microseconds to perform the arithmetic to produce each 16 bit result on a 1300MHz P4 processor with an 8KB primary cache and 256k secondary cache. Compiled as described in Table 1.

The effects of both these optimisations are shown in Tables 1 and 2. The broad features to note are :

- As vector lengths rise, the time taken to produce each byte of the result rises, this is due to generally poorer locality of access and thus generally poorer cache utilisation for larger vectors.
- In all cases the optimal code which evaluates temporary results in registers performs better than the naive code which allocates temporaries to memory buffers. In some cases it performs twice as well.
- The degree of advantage of the optimal code varies non-linearly with vector length as a result of interaction of the vector block sizes with the levels of the cache hierarchy. With certain combinations of sizes of vector and cache, the temporaries produced by the initial subtraction do not interfere with the source data. In these cases the performance penalty for the naive implementation is less.

Vectorisation of operations

It is desirable to make use of the vector instructions of current CPUs for array programming. Once an array expression has been re-organised by the compiler into the optimal form, it can be vectorised. Let us assume we are working with a CPU capable of performing operations on 4 numbers at a time. We can denote this in pseudo C as:

Algorithm 4: Vector version

```
// assume vector length 4
for(i=0;i<(zlen-3);i+=4)
  z[i:4]=a[i:4]*(b[i:4]-c[i:4]);
if( zlen %4)
 for(i=zlen-(zlen % 4);i<zlen;i++)
   z[i]=a[i]*(b[i]-c[i]);
```

Here the notation z[i:4] represents a sub-vector of length 4 starting at position i in z. Because we can not be sure that the vectors to be processed will all be divisible by 4 we need two loops. The first handles the vectorisable code, the second a remainder that has to be evalued in scalar fashion. Consider the case of a vector of length 8. The first algorithm will evolve as follows:

```
i=0
compute result for z[0:4]
i=4
i<5 yess
compute result for z[4:4]
i=8
i<5 no
terminate first loop, skip second
```

For a vector of length 6 the algorithm will evolve as:

```
i=0
compute result for z[0:4]
i=4
i<3  no
6%4 nonzero
i=(6-(6%4))=4
compute result for z[4]
i=5
compute result for z[5]
exit
```

If the length of the vectors is known at compile time, it is possible for the constant folding stage of the code generator to spot when the test:

```
if(zlen %4)
```

will return zero and delete both the test and the second loop from the code.

vector length	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}
optimal scalar μs	0.18	0.192	0.211	0.228	0.281	0.331
optimal vector μs	0.036	0.032	0.037	0.048	0.057	0.071
scalar/vector	4.74	6.0	5.7	4.75	4.93	4.66

Table 3: Comparison of scalar and vector code times for the statement used in the previous examples.

Use of vectorisation brings big performance benefits as is shown by Table 3. These are substantially greater than the speedups that can be obtained by the register optimisations discussed earlier. These sorts of optimisations are discussed in more detail in [2].

Implementation Issues

In the model advanced above, an array assignment writes directly into store that has already been allocated. In the statement x:=a*(b-c), the variable x is assumed to stand for an array whose rank is known at compile time, and whose size is either known at compile time, or has been determined at run time prior to the execution of the statement. Thus when performing a left to right parse of the statement, the parser knows rank of the destination before parsing the source expression.

This lends itself to a simple extension of recursive descent compiling techniques. Let us assume we have a procedure *expression* to parse expressions, which returns a semantic tree for the expression. After determining the rank r of the destination, the parser carries out the following steps:

1. Declare hidden loop iterator variables $\iota_0,\iota_1,...\iota_{r-1}$
2. Create an semantic tree *l* made up of r nested for loops using these iterators.
3. Let e= *expression*($[\iota_0,\iota_1,...\iota_{r-1}]$) to obtain a scalar semantic tree for the array expression.
4. Build the semantic tree of a scalar assignment of the form $x_{\iota 0.\iota 1,...\iota r-1}\leftarrow$ e and incorporate this into the loop nest *l*.

On encountering an array identifier the procedure *expression* attempts to reduce it to a scalar using the list of iterators passed to it. If the rank of the array is too large to allow its reduction, then a type error is signaled.

If instead of the parser being hand written, an automatically constructed bottom up parser is used in the compiler, an analogous algorithm can be performed whilst walking the syntax tree generated by the automatic parser.

Data Flow Issues

The previous section has argued against evaluating each binary array operator to produce a new array value, because it is more efficient to combine all subexpressions within a loop operating on a scalar expression. So long as one is dealing with vector or matrix expressions of the form $M\leftarrow A\Diamond B\Delta C$, where $\Diamond,\Delta$ are pairwise scalar operators, such as + or -, which are mapped over the matrices, no problems arise. The semantics of both implementation approaches are the same. When we allow other operators such a matrix transpose or matrix multiply, problems can arise.

Suppose we transpose the matrix

1 2

3 4

using the assignment $\mathbf{A} \leftarrow \mathbf{B}^T$, then **A** ends up with

1 3

2 4

and if we evaluated $\mathbf{B} \leftarrow \mathbf{B}^T$ we would expect **B** to also equal

1 3

2 4

However, if we have translated $X \leftarrow Y^T$ for 2 by 2 matrices *X,Y*, as:

```
for(i=0;i<2;i++)
 for(j=0;j<2;j++)
  X[i,j]=Y[j,i]
```

using this approach $\mathbf{B} \leftarrow \mathbf{B}^T$ would result in the corruption of B as the transpose was being evaluated giving

1 2

2 4

A similar corruption would occur with the evaluation of some matrix products by simple nested loops: A≤B.C would be safe but A≤B.A would not. It is to obviate these dangers that Fortran 90 defines the semantics of array assignments as having to evaluate the entire array valued expression on the right hand side of the assignment prior to performing the assignment itself. Vector Pascal takes the opposite approach, specifying that the semantics of array expressions is defined in terms of the equivalent loop construct with the compiler signaling an error if it encounters a data flow hazard in an array expression. Since, in the presence of parameter aliasing, it is not always possible to detect such data-flow hazards at compile time, one can argue that the Fortran 90 approach is safer. The additional cost incurred by the Fortran 90 semantics can be avoided in those cases where data-flow analysis can prove that no hazard exists - for instance where the arrays are not function parameters.

Array Bounds Checking

It is highly desirable that an array language have some system of array bounds checking to detect errors. The simplest way to do this is to plant explicit tests on array bounds whenever an array is indexed. Using C as an algorithmic notation this would take the general form:

```
if(i<lwb_a) boundsfault();
if(i>upb_a)boundsfault();
b=a[i];
```

which is clearly expensive and would dominate the cost of actual array access. But this can be done with as little as one extra instruction on an Intel processor using the

BOUND r, pair

instruction where `r` is a register and pair is reference to a two element vector in memory. This checks whether a register `r` is between the values `pair[0]`, and `pair[1]` generating an interrupt if it fails. Since a modern processor will be able to fetch 64 bits at time from memory, both bounds can be accessed in a single memory fetch, but the extra instruction and extra memory fetch can still represent an overhead of 50% or more on the original cost of array access. It is thus desirable to try and minimise the number of array bounds checks that are done.

Suppose we start with a loop with naive bounds checking:

```
for(i=x;i<=y;i++){
    if(i<lwb_a) boundsfault();
    if(i>upb_a) boundsfault();
    total+=a[i];
}
```

Then it is clear that $x \leq i \leq y$ throughout the loop. Thus if an array bounds fault is to occur, either $x < lwb_a$ or $y > upb_a$. We can safely move the bounds check to before the start of the loop:

```
if(x<lwb_a) boundsfault();
if(y>upb_a) boundsfault();
for(i=x;i<=y;i++)
 total+=a[i];
```

So long as x < y this formulation will have a lower overhead. Indeed in many case it allow the bounds check to be totally eliminated. If the array bounds and loop bounds are known at compile time, then the tests can be evaluated during compilation. If they pass the compiler can elide the bounds checking code, if they fail, it can signal a type error at compile time.

		code μs	ratio
optimal array bounds	vector	0.03	
naive array bounds	vector	0.036	
naive/optimal	vector		1.2
optimal array bounds	scalar	0.076	
naive array bounds	scalar	0.18	
naive/optimal	scalar		2.36

Table 4: Crusoe performance on x:=a*(b-c) in registers with no intermediate buffer assignment comparing naive with optimised array bounds checking, otherwise same test conditions as in tables 1 and 3.Vector lengths 1024.

In an array language, most of the *for* loops that the compiler deals with are automatically generated ones arising from array assignments. These lend themselves very well to the optimisations described above. For statically sized arrays, which are the predominant ones in classical Fortran or Pascal programming, most array bounds checks can be performed at compile time. For dynamically sized arrays most bounds checks can precede the loop. An exception comes in scatter-gather statements such as: `b:=a[c]`, where `c` is an array. In this case we must plant code to check that each `c[i]` is within the bounds of `a`.

As Table 4 shows, the propagation of array bounds checking outside of loops can make modest but very worthwhile improvement to performance for vectorised code, but makes a very big difference to performance on non-vectorised code. For the vectorised code, array bounds are checked for only 25% of the entries anyway, reducing the advantage of optimising the checks.

Hazards

With ordinary *for* loops certain hazards can arise with this sort of optimisation of array bounds checking. Consider the following Java code:

```
int [] a={1,2,4,8};
....
for (i=0;i<10;i++){
    if (i==0) a=new int[10];
    a[i]=getdata();
}
```

If we propagated the bounds check for a[i] to before the *for* loop thus:

```
if(a.length<9)throw new ArrayBoun
dsException(``a'');
for(i=0;i<10;i++){
    if (i==0) a=new int[10];
    a[i]=getdata();
}
```

An exception would be falsely signaled by the propagated bounds checking code. In general this hazard can arise with languages that have arrays implemented as pointers to heap data structures. In this case array bounds optimisations are only safe if data-flow analysis can prove that the array variable points to the same heap block throughout the loop. If arrays are static this problem obviously does not arise. Nor does it arise in array expressions, allowing these to be optimised more easily than standard for loops.

Implementation Issues

Condition	Truth value	Action
known at compile time	true	signal compile time error
known at compile time	false	do nothing
unknown at compile time		plant code for check

Table 5: Rules followed by plantcheck .

How can these bounds checking optimisations be implemented?

One simple approach is for the compiler to maintain a stack of currently active loop iterators as it processes the code. Associated with each loop iterator is an upper bound list and a lower bound list. The stack and loops are manipulated as shown in algorithm 5. When an array index operation a[i] is encountered algorithm 6 is invoked. The function plantcheck either follows the rules given in table 5.

Algorithm 5: to propagate bounds checks

```
push iterator i onto iterator stack

parse the body of the loop saving the gener-
ated code in buffer B

pop the iterator i from the stack

for each element e of the i.lwbList

  plantcheck(loopstart<e)

for each element e of the i.upbList

  plantcheck(loopfinish>e)

plant code in buffer B
```

Algorithm 6: to plant array bounds checking

```
CheckIndex(Array a, Expression i)
if i is an active loop iterator
 then
    append (i.lwbList,a.lwb)
    append (i.upbList,a.upb)
else
    plantcheck(i<a.lwb)
    plantcheck(i>a,upb)
```

Loop Unrolling

		code μs	ratio
unrolling	vector	0.018	
no unrolling	vector	0.028	
unrolling gain	vector		1.55
unrolling	scalar	0.49	
no unrolling	scalar	0.6	
unrolling gain	scalar		1.22

Table 6: Effects of use of loop unrolling, otherwise same test conditions as in tables 4 and 3.Vector lengths 1024.

A final optimisation that can be applied in a compiler is to unroll the loops. Suppose we have vectors that we know from type information is going to be of length 4, for example the quaternions used in graphics processing. In this case we can replace loops by the equivalent sequential code even in the absence of a vector instruction-set:

```
for(i=0;i<4;i++)z[i]=a[i]*(b[i]-
c[i]);
```

gets replaced by

```
z[0]=a[0]*(b[0]-c[0]);

z[i]=a[1]*(b[1]-c[1]);

z[2]=a[2]*(b[2]-c[2]);

z[3]=a[3]*(b[3]-c[3]);
```

The same principle can be extended to other loops whose iteration count is exactly divisible by some unrolling factor. The unrolling optimisation should be applied after vectorisation to get the best effect. Thus for arrays of length 1024 and a processor vector length of 4 we might have:

```
for(i=0;i<1021);i+=4){

    z[i:4]=a[i:4]*(b[i:4]-c[i:4]);i+=4;

    z[i:4]=a[i:4]*(b[i:4]-c[i:4]);i+=4;

    z[i:4]=a[i:4]*(b[i:4]-c[i:4]);i+=4;

    z[i:4]=a[i:4]*(b[i:4]-c[i:4]);
}
```

In the original un-vectorised loop, we had a conditional branch instruction for every word of the answer. In the vectorised and unrolled version we have one conditional branch per 16 words of the answer.

Aggregate effect

If we now consider the combined effects of all of the optimisations we can see that they speed array expressions up more than 10 times (table 7).

	improvement contribution	cumulative improvement
registers versus temporary array results	1.48	1.48
vectorisation	4.74	7.01
optimal array bound checks	1.2	8.41
Loop unrolling	1.55	13.0

Table 7: The cumulative effects on performance of all the array optimisations.

The gains from each optimisation combine multiplicatively. The exact figures given for each of the optimisations will depend on the machine it is being run on, the vector lengths and the efficiency of other aspects of the compilation process. In our examples we have used 16 bit arithmetic because this vectorises on any common-denominator Pentium compatible machine. The vectorisation gains for floating point code can be less.

Several of the optimisations are equally valid for array language interpreters. If these are partly written in assembler, then they can make use of vector instructions. Whether written in C or assembler they can unroll the inner loops of their primitive array arithmetic routines. Interpreters routinely eliminate array bound tests from their inner loops. A good interpreter can be competitive with compiled code. Table 8 shows comparisons between the A+ advanced APL interpreter and compiled Vector Pascal. Timings are given per floating point result for the calculation $x \leftarrow a \times b - c$ using vectors of length 1024. As before the non-pull-through version of the compiled code was implemented as two statements:

$$t \leftarrow b - c$$

$$x \leftarrow a \times t$$

to get round the fact that Vector Pascal uses pull-through by default. It can be seen that on both machines the interpretive code was competitive with the worst compiled code. Applying all of the optimisations described in this paper improves the speed of the compiled code by a factor of about 10. The peak performance of the compiled code is one useful floating point operation every two clock cycles, taking into account all the overheads of addressing 3 source and 1 destination arrays.

		750MHz Crusoe		1000MHz Athlon	
		µs	MFLOPS	µs	MFLOPS
a+ version	interpreted	0.097	20	0.036	56
no-pull-through	compiled	0.077	26	0.042	48
pull-through	compiled	0.047	42	0.026	76
pull-through plus loop unrolling	compiled	0.018	110	0.008	250
pull-through, loop unroll, vectorise	compiled	n/a		0.004	500

Table 8: Comparison of compiled versus interpreted loops. The interpretive timings are done using the A+ advanced APL interpreter. All compiled timings are for Vector Pascal. MFLOPs are calculated on the basis that there are 2 floating point operations per result.

Although for many purposes the flexibility of an interpretive solution will outweigh the speed of compiled code, for others such as graphics, speed is important. In these cases a compiled array language provides an effective solution, combining relatively concise notation with very high computational speed.

References

[1] Abrams, P.,An APL Machine, SLAC rep 114, Stanford Univ., 1970.

[2] Bik, A. J. C., Girkar, M., Grey, P. M., Tian, X., Automatic Intra-Register Vectorization for the Intel Architecture, International Journal of Parallel Programming, Vol 30, No. 2, 2002,pp. 65..97.

[3] Budd, T., An APL Compiler for a Vector Processor, ACM Transactions on Programming Languages and Systems, Vol 6, No. 3, July, 1984.

[4] Cockshott, P., Vector Pascal Reference Manual, ACM Sigplan Notices, Vol 37, pp 59-81, June 2002.

[5] Metcalf, M., and Reid., J., The F Programming Language, OUP, 1996.

[6] Snyder, L., A Programmer's Guide to ZPL, MIT Press, 1999.