

# Adding Intensionality to Functional Programs

E.A. Ashcroft and A.A. Faustini  
Arizona State University  
Tempe, Arizona

## Abstract

This paper presents the case for adding a new feature to functional programs, namely *intensionality*. The advantages of intensionality are simplicity, conciseness and implicit parallelism (and also the ability to talk about multidimensional objects in a functional framework). To illustrate the advantages, several programs are developed in IISWIM (Intensionally-extended ISWIM). One simple program for summing a vector is transformed and adapted until it gives us a mergesort program and a program to solve the Extended Hamming Problem (i.e., for an arbitrary number of primes). The programs are concise and elegant and, like many intensional programs, extend naturally to handle inputs of arbitrary dimensionality.

## Introduction

This paper will present a case for adding a new feature to functional programs, namely *intensionality*. Intensional programming [FaWa87] is based on a fundamental idea from intensional logic [Fro86, Tho74], namely that the meaning of an expression depends on some sort of implicit context. Intensional reasoning is used all the time by people in their everyday usage of natural language. For example, the phrase "the President of the United States" is usually associated with the person who happens to be in office when the cited phrase is used. However, in general there is no way of knowing which President is being referred to without knowing the context of usage of the phrase, i.e. the time and place of utterance of the phrase (the place might be necessary at some possible future time when there is a country called the United States of Europe). Since the context of usage is neither explicit nor implicit in the statement, the phrase seems to violate the basic principle of referential transparency, that the meaning of a whole phrase depends only on the meaning of its parts. Since referential transparency is the *sine qua non* of functional programming, adding intensionality to functional programs seems doomed to failure.

This problem can be resolved, however. If we say that an *extension* of an expression is the value of the expression in a given context, and that the *intension* of the expression gathers all these extensions together and captures how they depend on the contexts (i.e., the intension is a function from contexts to values), we can take the intensions of expressions as being their meanings, as far as referential transparency is concerned. The intension of an expression has the desired property that it depends only on the intensions of the parts of the expression.

Since adding intensionality to functional programs is no longer an intrinsically impossible endeavor, let us see what it might entail and why we might want to do it. The advantages of intensional programming all come from the conciseness and elegance of the programs and the high degree of parallelism implicit in the programs. Conventional functional programs are concise and elegant, but intensional functional programs are even more concise, as we will see. The implicit parallelism in conventional functional programs is not very high, being limited to the parallel evaluation of the operands of operators and defined functions. In intensional functional programs, we also have context parallelism; expressions can be simultaneously evaluated in many implicit contexts.

In order to add intensionality to functional programming, basically all we have to do is decide what is going to constitute the set of contexts, and what operators we will need that depend on contexts. Most of the operators, such as addition, will not depend on the context. Others will be able to look at their arguments in several contexts, not just the one in question. The ability to get from context to context is crucial, and is essentially the same as moving between possible worlds in modal logic.

The best way to show the possibilities of intensional programming is to look at example programs in a particular language, and we will consider Intensionally-extended (pure) ISWIM [Lan66]. We call it IISWIM. We could have extended a more modern language, such as Miranda [Tur84, PJ87], in basically the same way. We will show, for our first program, how we can get the same functionality from conventional (nonintensional) Miranda, but without, we fear, the conciseness and implicit parallelism that intensionality gives us.

### Contexts and Intensional Operators for IISWIM

Every variable or expression in IISWIM makes sense in some context, and each context refers to both "a point in time" and "a set of positions in particular dimensions (in space)". We will call these "the time context" and "the space context", though they are really components of contexts, not complete contexts. For concreteness, we will require that each member of a space context be of the form  $[i:j]$ , where  $i$  is the dimension (a natural number) and  $j$  is the position in that dimension (an integer). (Of course, two different members of a space context can not be for the same dimension.) It soon will become clear that when we refer to "time" we do not mean actual time, real time, absolute time or space-time. It is a formal concept, a direction orthogonal to space. In the same way, "space" is not real space. In particular, there need not be only three dimensions.

In order to define the intensional operators in IISWIM, we need to specify some context-producing meta-operators. For natural number  $i$  the meta-operator *inc- $i$*  maps contexts to contexts by simply incrementing the position in the  $i$ -th dimension by 1. Similarly, *dec- $i$*  decrements the  $i$ -th position. There are analogous operators for the time components of contexts: *for* moves the context forwards in time by one time step, and *back* moves it backwards by one time step. There are also meta-operators *modify*. For context  $C$  and integer  $j$ , *modify- $i$* ( $C, j$ ) denotes the context that is like  $C$  but with the member for the  $i$ -th dimension being  $[i:j]$ .



In the example IISWIM programs that we will present here, we need three generic binary intensional operators and one generic unary intensional operator. Each of the operators can be specialized to any particular dimension, or to time. The binary operators are *at*, *sby* (succeeded by), and *on\_finding*. The unary operator is *succ*. The specializations are *at\_t*, *at\_0*, *at\_1*, ..., *sby\_t*, *sby\_0*, *sby\_1*, ..., *succ\_t*, *succ\_0*, *succ\_1*, ..., and *on\_finding\_t*, *on\_finding\_0*, *on\_finding\_1*, ... , where the suffix *t* indicates time, and a particular numerical suffix *n* indicates dimension *n*. We also will need generic nullary intensional operators called *here*. They will be specialized in the same way as the others were.

The meanings of the operators are as follows. In all cases we will assume that we are given a context *C* and a natural number *i*. The meaning of the expression *a at\_i b* is the meaning of *a* in the context *modify-i(C, k)*, where *k* is the meaning of *b* in the context *C*. The definition of *at\_t* is analogous. The meaning of *a sby\_i b* depends on the position in dimension *i* according to the context *C*. If that position is 0, the meaning is the meaning of *a* in *C*. Otherwise the meaning is the meaning of *b* in context *dec-i(C)*. Again, the definition of *sby\_t* is analogous. The meaning of *succ\_i a* is simply the meaning of *a* in context *inc-i(C)*, and similarly for *succ\_t*. The meaning of *a onfinding\_i b* is a little more complicated. Basically, it involves incrementing through contexts to find the position at which *b* becomes true, and then returning the corresponding value of *a*. More precisely, its meaning is the meaning of *a* in the first context in the sequence *C*, *inc-i(C)*, *inc-i<sup>2</sup>(C)*, *inc-i<sup>3</sup>(C)*, ..., in which *b* has the meaning *true* (having previously been *false*). If no such context exists, the meaning is undefined, denoted  $\perp$ . The definition of *on\_finding\_t* is analogous. The meaning of *here\_i* is simply the position in dimension *i*, according to context *C*. The meaning of *here\_t* is simply the time context of *C*.

In an intensional language, the meaning of an expression depends on the context. In general, there will be an enormous number of possible contexts. In IISWIM, each context specifies a particular point in time and space. It is infeasible to compute the value of any expression at *all* contexts. One has to compute the values for the contexts that are needed by the program. An implementation of an intensional program seems necessarily to be demand-driven. Given that, it is not surprising that it is natural in an intensional program for objects (or intensions) to be potentially infinite. In the examples that we will consider, we will be summing up the first *n* elements of vectors. It is natural for the length of the vector to be unspecified, since only the values that need to be summed will be used or evaluated. The vectors are potentially infinite. In order to implement the same programs in a conventional language we will use potentially infinite lists.

### Program 1

The first program we will consider will be a program for finding the sum of a set of *n* numbers. A simple way of finding the sum is to go sequentially through the set keeping a running total, and then return that total after *n* steps. This conventional algorithm is easily coded in IISWIM. We will assume that the numbers are in a vector, in dimension 0, called *A*. Here is the program:

```
sum(A, n) = tot at_0 (n - 1)
  where rec
    n:-0;
    tot = A sby_0 tot + succ_0 A;
  end
```

(The declaration `n:-0;` indicates that the formal parameter `n` cannot vary in dimension 0. The syntax will be explained later.) The sum is defined in terms of a local vector `tot` of the running totals of `A`. The first element of `tot` is the first value of `A`, and each succeeding element is formed from the addition of the current value of `tot` and the next value of `A`. The definition

$$\text{tot} = A \text{ sby\_0 } \text{tot} + \text{succ\_0 } A;$$

conveys this by means of intensionality. The definition of `tot` is really specifying each element of the vector. It says that the element at position 0 is the value of `A` at position 0, and that at any position greater than 0, say  $i + 1$ , the value of `tot` is the value of `tot` at position  $i$  plus the value of `A` at position  $i + 1$ .

It should be clear that this explanation of how the various values in the vector `tot` are defined in terms of each other implies a definite order in which they have to be calculated. In order to get `tot` at position  $i + 1$  we need `tot` at position  $i$ . The values must be calculated in a sequential order. But it is important to realise that this order is not expressed in the program in an explicit or imperative way by means of some sort of control structure. It is implicit in the way that the values in the vector `tot` depend on each other, that is, in the so-called data dependencies. There is a sort of implicit loop in the program.

Continuing with the examination of the program, it should be clear that the value of `tot` at position  $k-1$  will be the sum of the values of `A` from position 0 through position  $k-1$ , that is, the first  $k$  values of `A`. Thus, if  $n$  is the value of `n` in the original context, the desired value, namely the sum of the first  $n$  values of `A`, is specified by the expression

$$\text{tot at\_0 } (n - 1)$$

There is no explicit termination condition for the implicit loop. The program simply extracts from the loop the value of `tot` that it wants. The implicit loop has an implicit termination condition: Any sensible implementation of IISWIM will only compute values that are wanted (or perhaps a few more, if that can improve the effectiveness of the implementation in some way), so once the desired value has been extracted no more computation will take place.

For Program 1 we have not talked at all about contexts that refer to "points in time". Therefore, everything we have said *applies to any point in time*. Which means that if `A` is a different vector at each point in time, and `n` is a different (scalar) number at each point in time, the program, *as written*, will, at each time point, sum up the first `n` elements of the vector `A` (for that time point). Each of the vectors is summed up sequentially, as explained above, but there is no implicit order in which the different vec-



tors have to be summed. As we mentioned before, the use of the word "time" should not be taken literally; it is just a formal notion. Here, the computations for different "time"s could take place simultaneously in a parallel implementation. Given an arbitrary number of vectors, all of length  $k$ , say, a suitable parallel implementation of IISWIM could sum them all up in parallel in  $k$  steps, without there having to be an explicit command to do so and without there having to be any analysis of the program to detect the parallelism. This is an example of the implicit parallelism in intensional programs and an indication of how it could be exploited.

Similar arguments can be made about "positions in space". In this program we have been considering the variables  $A$  and  $\text{tot}$  to be vectors in dimension 0, that is, objects that depend on space dimension 0. The variable  $n$  does *not* depend on dimension 0. (This ensures that the value of  $\text{tot}$  at  $\_0$   $n$ , and hence the value of the program, does not depend on dimension 0, also.) The variables  $A$  and  $n$  do not need to be just vectors or scalars, they could have much higher dimensionality than that. Before explaining that statement, we must explain what we mean by "dimensionality" or "rank".

The dimensionality of an object (i.e. of an intension) is the number of different dimensions occurring in space contexts for that object. There are syntactic restrictions we can place on IISWIM programs that ensure that an object has the same dimensionality at different points in time, and that that dimensionality does not vary in space, either (that is, the elements of a vector, or array, etc., have the same dimensionality). All the programs considered here satisfy these restrictions.

In this paper we will say that the rank or dimensionality of an object basically tells us which dimensions the object *cannot* vary in and the minimum set of dimensions it *should* vary in. Also, the rank will tell us whether the object should or cannot vary in time. As we will see, objects will be able to vary in more dimensions than the minimal ones that are specified in their rank, and the program in which the object is used will "extend" naturally to the extra dimensions. Rank is specified by two disjoint sets of dimension numbers together with an indication whether the object should or cannot vary in time. It is convenient to represent rank by means of a string of 0's and 1's and -'s. The first symbol will be for time and the rest for space dimensions. For example, if an object might vary in time, cannot vary in dimensions 1 and 3, and should vary in dimension 4, its rank will be --0-01. The first - indicates that it might vary in time, the 1 indicates that it should vary in dimension 4, and the 0's indicate that it cannot vary in dimensions 1 and 3. The other -'s indicate that it *might* vary in dimensions 0 and 2. It also might vary in any of the dimensions not mentioned, i.e., dimensions 5, 6, etc. Note that a scalar that cannot vary in time will have rank 000000000..., an infinite sequence of 0's. We will denote this by 0\*. For reasons of computability, the rank of an object can have only a finite number of 1's.

Note that for this program  $n$  has rank 0 and  $A$  has rank 1. Both  $A$  and  $n$  potentially can vary in dimensions 1, 2, 3, etc. No declaration is needed to proclaim this, and, in fact, it is generally difficult to prevent a variable from potentially varying in an arbitrary number of dimensions. Which dimensions a variable does vary in can only be answered for a particular run of the program for particular input, be-

cause it depends, ultimately, on the actual values of the input variables.

Which dimensions an input variable or formal parameter cannot vary in can not be determined from the definitions in the program; it has to be specified in a "declaration". The declaration is by means of a string of 0's, 1's, and -'s; only the positions of the 0's is important. Which dimensions the input variables should vary in *can* be determined from the program itself. In this example, the use of the operators `sby0` and `at_0` is enough to indicate that `A` should vary in dimension 0.

Now, for this program, exactly the same argument we made about time can be made with respect to space dimensions 1, 2, 3, etc. If `A` were three dimensional, say, we could think of the object as a matrix (in dimensions 1 and 2) of vectors in dimension 0. Program 1, *without change*, would give a matrix (in dimensions 1 and 2) of the sums of the vectors, provided `n` were a matrix (in dimensions 1 and 2) of scalars, the lengths of the vectors. These sums would be calculated in parallel by an "intensionally exploitative" architecture.

This way in which the program "extends" to higher dimensions is a general property of IISWIM. As long as the inputs to a program vary in the dimensions that they "should" vary in and don't vary at all in those they "cannot" vary in, the program will extend uniformly to any number of other dimensions, and to time too, if the program doesn't mention time. It may be interesting to see how this extensibility can be incorporated into the same program written in a language similar to Miranda.

In order to get the same functionality, we will assume that the parameter or input `A` is a potentially infinite list, and we will define a potentially infinite list `tot`, exactly as in IISWIM. If we ignore the extensibility for the moment, the only problem is that we must define a pointwise addition operator for such potentially infinite lists. Here is the Miranda-like program:

```
sum A n = tot!(n-1)
  where rec
    tot = hd A : add tot (tl A)
    add a:x b:y = (a + b): add x y
```

In order to get the extensibility we must code up multidimensional arrays as lists of lists. (In fact, we should also consider *time-varying* multidimensional arrays but this would introduce a complication that we need not consider here.) The IISWIM program handles arguments of arbitrary dimensionality. The Miranda-like program we give here can handle two-dimensional arguments. We have attempted to construct a program that is equivalent to the IISWIM program for arguments of all dimensions, but have not yet been able to do so.



```
sum A n = bang tot (n-1)
  where rec
    tot = hd A : add tot (tl A)
    add a:x b:y = (a + b): add x y
    add a b:y = (add a b):(add a y)
    add a:x b = (add a b):(add x b)
    add a b = a + b
    bang L ((a:x):y) = (bang L a:x):(bang L y)
    bang L b:y = hd(bang L b):(bang (mapcar L tl) y)
    bang L b = L!b
```

The function **bang** is an attempt to extend the indexing operation of Miranda so that it works like **at\_0**.

## Program 2

Even more implicit parallelism is available for exploitation in a suitable architecture if we solve the summing problem in a different way. The recursive “divide and conquer” approach to summing says that to sum a set of numbers we just need to split the set into two roughly equal parts, sum each of the parts, and then add the two results. The recursion can be seen to be building a binary tree of addition operators, with these operators waiting for results to come flowing back when the recursion stops. These results start to come from the elements to be added, which will form the leaves of the tree. Knowing that the recursion algorithm would build this binary tree, we can instead start with the tree and proceed iteratively, from the leaves of the tree. We can see how the algorithm would proceed: The sum of a vector of length  $2^k$  is obtained by first adding together the elements of the vector in pairs in parallel, giving a new vector, then summing pairs of the new vector in parallel, giving yet another new vector, then summing pairs of that new vector in parallel, and so on. In  $k$  steps of parallel computation we will have the sum of the  $2^k$  elements. Notice that this description implies that each new vector will be half the size of the previous one, and that each element, at position  $i$ , say, in one of the new vectors, is the sum of the two elements in the previous vector that were at positions  $2*i$  and  $2*i+1$ . See Figure 1 which conveys this idea graphically. This technique can be implemented in IISWIM as follows:

```
sum(A, n) = new at_0 power_of_two
  where rec
    n:-00;
    new = A sby_1 (new + succ_1 new) at_1 (2*here_1);
    power_of_two = here_0 on_finding_0 2**here_0 >= n;
  end
```

This program sums up the elements of a vector  $A$  (in dimension 1 this time) whose length,  $n$ , is a power of two. ( $n$  cannot depend on dimension 0 or dimension 1.) The variable `power_of_two` is defined to be that power. (`power_of_two` is defined to be the integer corresponding to the first position  $i$  in dimension 0 at which  $2^i$  is found to be no less than  $n$ .)

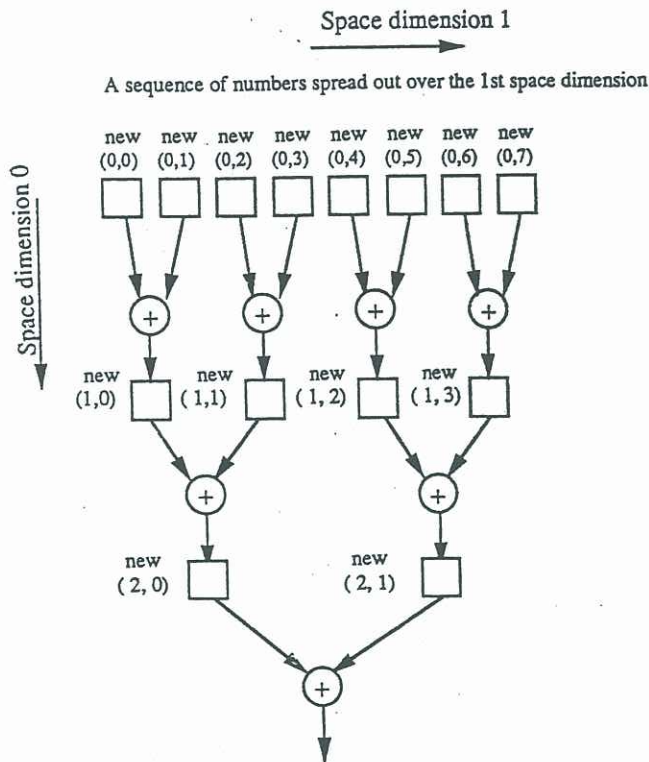


Figure 1 : Summing a vector whose length is a power of two



This program explicitly uses two space dimensions. The program requires a variable `new` that denotes the vector of the new vectors mentioned in the earlier description of this parallel programming technique. The variable `new` is two dimensional, with the first dimension (dimension 0) indicating which new vector we are talking about and the second dimension (dimension 1) indicating which element of that vector we are considering. The statement

```
new = A sby_1 (new + succ_1 new) at_1 (2*here_1);
```

defines `new` appropriately. It takes the rank -01 object `A` and produces the rank -11 object `new`. The first vector of `new` is simply `A`. Any succeeding vector, say the  $t+1$ th, will consist of elements that are defined in terms of the elements of the  $t$ th vector. In fact, the definition of `new` tells us that the element in the context  $\{[0:t+1], [1:s]\}$  will be the value of `new + succ_1 new` in context  $\{[0:t], [1:2*s]\}$ . This will be the sum of the value of `new` in that same space context and the value of `new` in space context  $\{[0:t], [1:2*s+1]\}$ . In other words, we are adding together exactly the elements in the previous vector that we were supposed to.

Notice that all the elements of the new vector can be computed simultaneously. There is no construct in the language to express this fact. In an intensional program, for every variable we are always defining *every* value of the variable in *every* context in one definition. It is only data dependencies such as occurred in Program 1 that prevent the simultaneous computation of these values. *In intensional programs, parallelism is the norm.*

The correct value is obtained for the sum of vector `A` by extracting the value of `new` at the dimension 0 position corresponding to `power_of_two` (using `at_0`). The value of the `at_0` expression is a vector, of rank -001 (because `power_of_two` has rank -001, the same as the rank of `n`).

Program 2, like Program 1, does not pay any attention to time contexts, and, though it pays attention to dimension 1 space components, it does not pay attention to dimensions higher than that. It therefore will extend uniformly to arbitrary times and to space dimensions higher than 1. In a suitable architecture, it will perform very well. Arrays of  $n$ -element vectors will be summed in  $\log_2 n$  parallel steps, independent of the number of arrays (at different times) or the ranks of the arrays! The "suitable architecture" must have massive amounts of parallel processing capability. The Connection Machine springs to mind. (This is not just a glib remark. Implementation of IISWIM on the Connection Machine is being seriously considered.)

Unfortunately, Program 2 can only sum up vectors of lengths equal to powers of two. The following modification will rectify this problem.

### Program 3

One obvious way to handle vectors of lengths that are not powers of two is to "pad them out" with zero's. This causes wasteful computation, however, since there will be many additions of zero to zero. Other modifications, that involve testing values before they are added, to ensure that they are legitimate,

have the unfortunate consequence of reducing the parallelism in the program, perhaps reducing the program to a sequential one. The following modification, due to Alan Jorgensen, who is studying at ASU, avoids both these problems. It prunes the tree using a numerical test on the width of the tree.

```

sum(A, n) = new at_0 power_of_two
  where rec
    n:-00;
    new = A sby_0 (if width mod 2 eq 0 then new + succ_1 new
                  else (new sby_1 new + succ_1 new) fi
                  at_1 (2*here_1));
    width = n sby_0 ceil(n/2);
    power_of_two = here_0 on_finding_0 width eq 1;
  end

```

Figure 2 shows how this program modifies the tree.

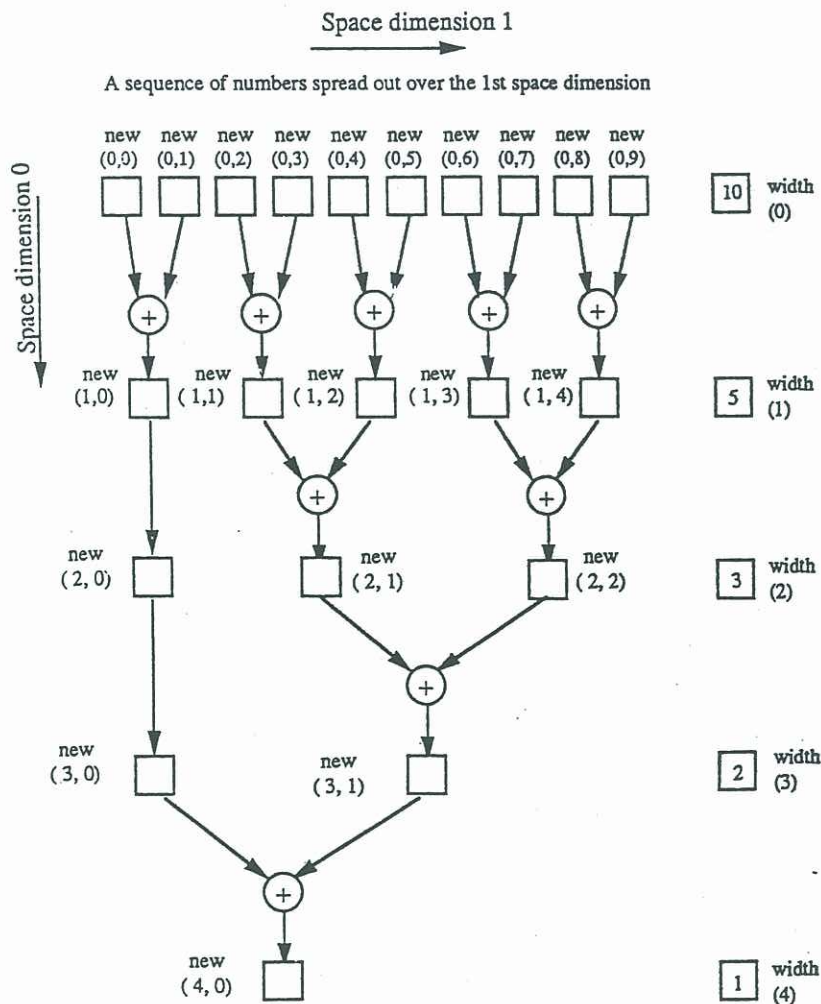


Figure 2 : Summing a vector whose length is not a power of two



#### Program 4

Program 3 essentially is based on a divide and conquer technique, as explained earlier. The technique is useful for other things as well as summing. For example, one can use it to find the maximum or minimum element in a vector. It is almost trivial to change the IISWIM programs above to accomplish these new tasks. All that is required is that “+” be replaced by “max”, or by “min”. Clearly, what is really called for is a higher order function that can be given any binary, associative function. Here is the code:

```
reduce(A, n, op) = new at_0 power_of_two
  where rec
    n:-00;
    new = A sby_0 (if width mod 2 eq 0 then op(new, succ_1 new)
               else (new sby_1 op(new, succ_1 new) fi
               at_1 (2*here_1));
    width = n sby_0 ceil(n/2);
    power_of_two = here_0 on_finding_0 width eq 1;
  end
```

This program can be used with any binary, associative operation, as long as its implementation does not depend on, or affect, the dimension 0 or 1 space contexts. In fact, we can even use a function or operation that depends on *time* as long as it is associative and doesn't impinge upon dimensions 0 and 1. One such function is the merging of two monotonically increasing time sequences. This example springs to mind because “mergesort” is another divide and conquer algorithm, and its binary tree is a binary tree of such merge functions, with the leaves of the tree being short time sequences, each containing just one of the elements to be sorted.

We can easily give a recursive definition of the merge function, as follows

```
merge(a, b) = if (a <= b) at_t 0 then a sby_t merge(succ_t a, b)
              else b sby_t merge(a, succ_t b) fi;
```

It is far better to use a nonrecursive definition, as follows:

```
merge(x, y) = if xx <= yy then xx else yy fi
  where
    xx = advanced_upon(x, xx <= yy);
    yy = advanced_upon(y, yy < xx);
    advanced_upon(a, b) = a at_t t1
      where rec
        t1 = 0 sby_t if b then t1 + 1 else t1 fi;
      end;
  end;
```

However we get a merge function, we can use it to define a mergesort program as follows:

```
_mergesort(L, n) = reduce(L sby_t maxnum, n, merge)
```

The constant **maxnum** is intended to be larger than any number in the list **L**. In order for the expression

```
L sby_t maxnum
```

to work, **L** must be constant in time, having rank 001. The expression gives a vector of short time sequences, of rank 101. These short time sequences are merged, getting longer and longer until the final sorted sequence is produced. Only the first **n** elements of that sequence should be examined; the **n** elements after that are all equal to **maxnum**.

On a suitably parallel architecture, with enough processors, the program will sort in linear time. And, of course, it extends to dimensions 2, 3, etc. (**L** can be an array of vectors, for example, and it will sort each of them.)

It is clear that the program will sort in linear time because at each parallel step one sequence value will move down at each level in the tree to the next lower level and, in particular, the value moving down from the lowest level is the next value in the sorted sequence.

It is worth pointing out that any implementation of IISWIM must compute Program 4 by having the various merge functions at different levels on the tree executing simultaneously, in a sort of producer/consumer fashion. This is generally regarded as a sophisticated parallel algorithm, but here it will happen automatically.

## Program 5

Hamming's Problem is the problem of producing in increasing order without duplications the sequence of positive integers of the form  $2^i 3^j 5^k$ , for any natural numbers  $i, j$ , and  $k$ . Hamming's Problem can be solved very elegantly in IISWIM. Here is such a solution:



```
h
  where
    h      = 1 fby merge(merge(2*h, 3*h), 5*h);
    merge(x, y) = if xx <= yy then xx else yy fi
                where
                  xx = advanced_upon(x, xx <= yy);
                  yy = advanced_upon(y, yy <= xx);
                end;
    advanced_upon(a, b) = a at_t t1
                where rec
                  t1 = 0 sby_t if b then t1 + 1 else t1 fi;
                end;
  end
```

(Note that this program doesn't use exactly the merge function given in Program 4. This one removes duplicates, but for sorting we wanted the duplicates preserved.)

An interesting extension to the problem is to remove the restriction to the set {2, 3, 5} in order to ask if the problem can be solved neatly for any set of primes. (It has been suggested by John Feo of Lawrence Livermore Laboratory that such a programming task would "test a language's ability to express recursive stream computations and to handle an unknown number of tasks".)

Basically, the IISWIM program above is merging three streams, using two applications of the function merge. The three streams are just the stream **h** that is being produced, multiplied in a pointwise fashion by each of the primes, 2, 3, and 5.

It is apparent that the extended Hamming Problem would be simple if we merge an arbitrary number of streams (just as we do in Program 4), each one formed by multiplying the stream being produced by one of the primes. If the primes form a vector in space, this production of a vector of streams is simple: we just say **primes\*h**.

We therefore can combine the above Hamming program with the mergesort program, as shown here:

```
hamming(primes) = h
  where
    primes:001;
    h      = 1 sby_t reduce(primes*h, n, merge);
    n      = here_1 on_finding_1 iseod primes;
    merge(a,b) = if aa <= bb then aa else bb fi
              where
                aa = advanced_upon(a, aa <= bb);
                bb = advanced_upon(b, bb <= aa);
              end;
    advanced_upon(a, b) = a at_t t1
      where rec
        t1 = 0 sby_t if b then t1 + 1 else t1 fi;
      end;
  end
```

The test `iseod` ("is end of data") enables `n` to be defined as the number of primes in the vector; we don't need to know the number of primes ahead of time.

The program, as expected, computes the Hamming Numbers for the extended problem. The interactions between the time computations and the space computations are quite subtle, but, because the program was conceived and programmed at a very high level, the details of the interactions in an actual implementation are interesting to observe, after the fact, but were not a cause for concern or trepidation during the actual writing of the program.

As with the mergesort program, the various activations of the merge function will run in parallel (pseudo or actual). Since the program never terminates, but continually produces output, its performance must be measured in terms of time steps taken per output produced. If there are  $k$  primes, an implementation on a parallel machine would produce each Hamming number after at most  $\log_2 k$  parallel steps. It is anticipated that many less than  $\log_2 k$  steps would be required on average, but a proper analysis would have to be based upon (perhaps yet undiscovered) statistical properties of Hamming Numbers. The worst case is when the next Hamming Number to be produced is to be formed from some prime multiple of the last one that was produced. That will require  $\log_2 k$  steps, to get the multiple through the tree. But that situation should be relatively uncommon.

These last two programs (for mergesort and the extended Hamming Problem), by their very similarity, illustrate how much the parallelism analysis can depend on the input data for the program in question. To some extent, the difficulty of analysis of the programs could have been anticipated just from the occurrence of the function `advanced_upon`, because `advanced_upon` requires testing of its input. Programs such as those for vision applications, for example, which do not involve any such testing, are much easier to analyze, and could be analyzed automatically.



## Conclusion

We have illustrated that intensional programs are concise and highly expressive. The programs are referentially transparent and enjoy all of the properties that make functional programs so desirable. In addition, they extend naturally to arguments of arbitrary dimensionality and have a high degree of implicit parallelism of which advantage can be taken without a great deal of program analysis being required.

## Acknowledgments

The work reported here was supported, mainly, by NSF Grants DCR-84 15618 and DCR-84 13727

## References

- [FaWa87]      Faustini, A.A., and Wadge, W.W.  
                 Intensional Programming  
                 The Role of Languages in Problem Solving 2, Editors J.C. Boudreaux et al  
                 Elsevier Science Publishers (North Holland), 1987.
  
- [Fro86]        Frost, Richard  
                 Introduction to Knowledge Base Systems  
                 MacMillan, Publishing, New York,  
                 1986.
  
- [Lan66]        Landin, P.J.  
                 The Next 700 Programming Languages  
                 CACM 9, No 3, March 1966
  
- [PJ87]         Peyton Jones, S.L.  
                 The Implementation of Functional Programming Languages  
                 Prentice Hall International  
                 1987
  
- [Tho74]        Thompson, R., editor  
                 Formal Philosophy, Selected Papers of R. Montague, Yale University  
                 Press, New Haven, Conn., 1974
  
- [Tur84]        Turner, D. A.  
                 Functional Programs as Executable Specifications,  
                 Phil. Trans. R. Soc. London, A312, pp. 363-388.  
                 1984