Programming Generality and Parallel Computers

Computation Structures Group Memo 287
May 1988

Arvind, S.K. Heller, and R.S. Nikhil

Also appears in Fourth International Symposium on Biological and Artificial Intelligence Systems, Trento, Italy, September 18-22, 1988.

This report describes research done at the Laboratory of Computer Science of the Massachusetts Institute of Technology. Funding for the Laboratory is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-84-K-0099.

Programming Generality and Parallel Computers

Arvind, Steve Heller and Rishiyur S. Nikhil

MIT Laboratory for Computer Science

May 27, 1988

Abstract

In recent years, many "parallel, general-purpose" computers have become available, using various interconnections of processors and memories. However, extensions to conventional languages to program these machines have made programming significantly more complex— a clear regression in software technology. The user now has the additional problem of partitioning his program into parallel parts, and often this is very machine-specific. Further, it is no longer easy to write determinate programs, making debugging a nightmare. In this paper we present an alternative approach. Id is a declarative, implicitly parallel language that simultaneously raises the level of programming and reveals much more parallelism than is possible with programmer annotations. Determinacy is guaranteed by the language semantics. We demonstrate this by developing a program in Id for a non-trivial problem called the "paraffins problem", and examining the available parallelism, after the fact.

1 Introduction

It is clear that parallelism will be fundamental to computing in the future. In recent years, many "parallel, general-purpose" computers have become available, using various interconnections of conventional processors and memories. These machines are usually programmed in conventional sequential languages (such as FORTRAN, C or Lisp) that have been extended with constructs or annotations by which the programmer indicates that certain things may be done in parallel. These extensions are necessary because the automatic detection of adequate parallelism from sequential programs remains a difficult problem, in spite of recent advances in compiler technology.

If anything, these parallel annotations constitute a regression in the level of programming [10]. The programmer now has the additional task of deciding how to partition his program into parallel parts. In some cases, it is easy to identify what can be done in parallel— for example, matrix multiplication has n^3 unrelated, and hence parallel multiplications. But even then, there is usually no clear criterion as to how the partitioning should be done (should all n^3 multiplications be done in parallel, or should only n^2 inner-products be done in parallel?) Usually, the decision is based on machine-specific parameters such as number of processors, task-creation overhead, synchronization overhead, network topology, *etc.*

In many (if not most) other cases, where the parallelism is less structured, it may be difficult to identify and code the parallelism in an algorithm. For example, it is non-trivial for the programmer

[0]This research was done at the MIT Laboratory for Computer Science. Funding for this project is provided in part by the Advanced Research Projects Agency of the Department of Defense under the Office of Naval Research contract N00014-84-K-0099.

Address for all three authors:
MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.
ARPANET addresses: arvind@xx.lcs.mit.edu, heller@xx.lcs.mit.edu, nikhil@xx.lcs.mit.edu

to understand just how much parallelism there is, say, in LU decomposition. Or, consider a vector product in which one of the input vectors is itself the result of another vector product. There is much producer-consumer parallelism here, but it is usually not exploited in languages with explicit parallelism. Whenever this dynamic behavior is difficult to conceptualize and reason about, the programmer avoids coding it. Explicit coding of such parallelism often makes the program non-modular; for example, the programmer has to code the two vector products together, rather than calling an existing vector product routine twice.

There is something very strange here. In all these cases, the *algorithms*, mathematically expressed, are abundantly parallel. However, when coded into a sequential language, *all* the parallelism is discarded, and the order in which operations are to be performed is overspecified to the limit. Now, the programmer (or the compiler) has to *reintroduce* parallelism into the program, using annotations. The difficulty in doing this biases our thinking to the point where is a widespread belief that existing algorithms may not have adequate parallelism, that parallel machines will be utilized effectively only by designing new algorithms.

Another serious issue is the potential loss of determinacy. Even though matrix multiplication is conceptually a determinate computation, if the programmer carelessly used the same variable to accumulate two inner-products computed in parallel, he is likely to obtain indeterminate behavior. Such mistakes are very easy to make, and it is notoriously difficult to detect them. Further, indeterminate behavior is difficult to reproduce, for debugging, because it can depend on input data and machine-specifics such as configuration and load, and even on the fact that the debugger is active.

There is already a "software crisis" with sequential machines; writing, debugging and maintaining large, complex programs is very difficult. As we move to parallel machines with significantly improved performance, we are only going to become more ambitious in the problems we tackle. Consequently, rather than complicate the programming process, it is even more urgent that we *raise* the level of programming.

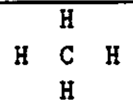
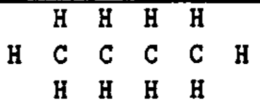
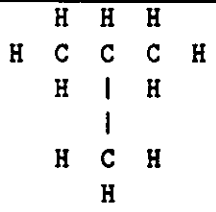
We recommend an alternative approach to parallel programming. The programmer specifies an algorithm in Id, a high-level, declarative language [15]. The programmer does not explicitly encode parallelism; instead, it is implicit in Id's operational semantics. *None* of the conceptual parallelism in the algorithm is obscured. Of course, the implicit parallelism comes with determinate semantics—a program is indeterminate only if it is deliberately introduced by the programmer for genuinely non-deterministic computations (*e.g.,* transactions against a shared database).

In Id, parallel components compose in a modular fashion, *i.e.,* producer-consumer parallelism is exploited automatically. Our experiments show that there is plenty of parallelism, *even in existing algorithms*. In other words, while the development of new algorithms is always a worthwhile and exciting effort, we cannot lay all the blame for a disappointing lack of speedup on existing algorithms.

In this paper, we demonstrate our approach by developing a program for the non-trivial "Paraffins Problem" (which we explain in the next section). We will concentrate on producing a clear, high-level program, without any conscious consideration of parallelism. Efficiency issues are tackled at the algorithmic level and not at the level of managing resources in some parallel machine. We will then show experimental results that quantify the parallelism in our solution. Finally, we comment on the difficulty of coding this problem without Id's high-level abstractions.

2 The Paraffins Problem

The problem we attack is one that was posed and solved by David Turner in [26]. Paraffins are molecules with chemical formula C_nH_{2n+2}, where C and H stand for carbon and hydrogen atoms, respectively, and $n > 0$. Carbon and Hydrogen atoms have valence 4 and 1, respectively. Here are some examples of paraffins:

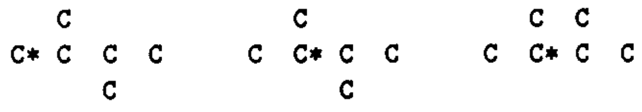
Methane (CH_4)	Butane (C_4H_{10})	Iso-Butane (C_4H_{10})
		

The butane example shows that for a given n, there can be many *isomers*, *i.e.*, distinct paraffins with the same formula C_nH_{2n+2}.

We can describe the structure of paraffin molecules in terms of *radicals*. Let us pick any carbon in a paraffin molecule, call it the *root*— its four bonds are each attached to a radical, which is a molecule of valence 1, with formula C_nH_{2n+1}. The structure of a radical can be defined inductively as follows:

- A hydrogen atom is a radical.
- If $r1$, $r2$ and $r3$ are three radicals, then attaching them to three bonds of a carbon atom— `(C r1 r2 r3)`— also constitutes a radical.

Note that this description is not unique. First, the four bonds of a carbon atom are chemically indistinguishable, so that `(C r1 r2 r3)`, `(C r2 r1 r3)`, `(C r2 r3 r1)`, ... are all the same radical. Second, the choice of the root carbon in a paraffin is arbitrary, so that, for example, `C` attached to four radicals `CH`$_3$, `CH`$_3$, `H`, and `H` is the same paraffin as `C` attached to `C`$_2$`H`$_5$, `H`, `H`, and `H`.[1] Here are some examples of different pictorial representations of the *same* molecule (by convention, we omit the hydrogen atoms, showing only the carbon "spine", and place an asterisk to the right of the chosen root):



It may interest the reader to note that for this particular `C`$_6$`H`$_{14}$ molecule, there are 108 equivalent representations.

The specification of the programming problem is this:

"Generate all distinct paraffins containing up to n carbon atoms."

What makes makes this problem especially interesting is that we do not want the result to contain multiple representations of the same paraffin. A straightforward solution would be first to pick a representation for paraffins and enumerate all possibilities containing up to n carbons. Then, we

[1] Radicals are called *oriented*, or *unordered trees* and paraffins are called *free trees*; see [11] for a general discussion of these structures, including the equivalence issue.

could group them according to the equivalences described above, and pick just one member from each group. In fact, Turner's original solution was coded in this way.

In this paper, we develop a solution that is more efficient in that it never generates any paraffin that does not belong in the final output. The basic idea is this. We define a "canonical" form for paraffins such that there is exactly one in each equivalence class. Then, our algorithm will only generate these canonical representatives.

As we develop the solution, we will pay *no* attention at all to parallelism issues. In the end, we will examine how much parallelism our solution actually contains.

3 Radicals, Orderings and Canonical Forms

The inductive definition of radicals can be expressed directly in Id:

```
type radical = H | C radical radical radical ;
```

This declaration introduces a new *data type* called `radical`. It is an example of a *union type* with two disjuncts, *i.e.,* objects of type `radical` are either the constant `H`, or a data structure `(C r1 r2 r3)`, where the three components `r1`, `r2` and `r3` are themselves objects representing radicals. The constants `H` and `C` are called *constructors*, and can be viewed as a "tag" that identifies the data structure.[2]

Let us define a function `carbons` that counts the number of carbon atoms contained in a radical (we shall also speak of the number of carbons in a molecule as its *size*):

```
def carbons H             = 0
  | carbons (C r1 r2 r3) = 1 + (carbons r1) + (carbons r2) + (carbons r3) ;
```

which reads as follows: If the argument to the function `carbons` matches the *pattern* `H`, the count is 0. If the argument matches the pattern `(C r1 r2 r3)`, the count is 1 (for the carbon) plus the sum of the carbons in `r1`, `r2` and `r3`. As in the second clause, a pattern can also introduce names (here, `r1`, `r2` and `r3`) for the corresponding components of the actual argument, and these names may be used on the right-hand-side of the clause. The clauses of a definition are not ordered; we could just as well have written:

```
def carbons (C r1 r2 r3) = 1 + (carbons r1) + (carbons r2) + (carbons r3)
  | carbons H             = 0 ;
```

The patterns must be mutually exclusive (this is checked by the compiler).[3]

We can define a canonical form for radicals, assuming we had a total ordering on radicals:

- The radical `H` is in canonical form.
- A radical `C r1 r2 r3` is in canonical form if and only if `r1` $\leq$ `r2` $\leq$ `r3`, *and* `r1`, `r2` and `r3` are themselves in canonical form.

[2]Union types are analogous to "variant records" in Pascal, except that (a) our variants *always* have discriminating tags (here H and C), and (b) components of a record (here, three radicals) are accessed by position, not by name.

[3]Although the clauses with patterns in an Id definition are reminiscent of clauses in a logic programming language like Prolog, the similarity is only superficial. In functional languages, there are no "logic variables"; pattern-matching is completely deterministic and does not require unification.

How do we define this total ordering? First, a comparison of two radicals will result in r1 < r2, r1 = r2 or r1 > r2. We define a new type containing three constants to represent these outcomes:

```
type ordering_outcome = lt | eq | gt ;
```

This is, again, a union type, with three disjuncts.[4]

Here is a function called ordering that computes the relative ordering of any two radicals, based on the number of carbons they contain:

```
def ordering  H               H              = eq
  | ordering  H               (C r21 r22 r23) = lt
  | ordering (C r11 r12 r13)  H              = gt
  | ordering (C r11 r12 r13) (C r21 r22 r23) =
          { nc1 = carbons (C r11 r12 r13) ;
            nc2 = carbons (C r21 r22 r23)
          In
            if      (nc1 < nc2) then lt
            else if (nc1 > nc2) then gt
            else {case (ordering r11 r21) of
                      lt = lt
                    | gt = gt
                    | eq = {case (ordering r12 r22) of
                              lt = lt
                            | gt = gt
                            | eq = (ordering r13 r23) }}} ;
```

The first three clauses are straightforward: comparing a hydrogen with a hydrogen, a hydrogen with a non-hydrogen and *vice versa*. The fourth clause compares two non-hydrogens, and is written as a *block*. A block introduces a nested scope (analogous to BEGIN ... END in Pascal), in which there are some declarations specifying values for local variables, followed by the keyword In, followed by the *body* of the block, which is an expression representing the final value of the block. Here, two local declarations bind variables nc1 and nc2 to sizes of the given radicals, which are computed using the carbons function. In the body of the block, the first two lines of the conditional expression specify that if sizes are unequal, we immediately know the ordering. In the final else clause, the sizes are known to be equal. The outer case expression recursively determines the ordering of the corresponding component radicals r11 and r21. If lt or gt, the ordering is known. If equal, the nested case expression similarly tests the ordering of r12 and r21. If these, too, are equal, the ordering is determined by that of r13 and r23.

The case expression is a fundamental construct in Id, and is used to dispatch on the disjuncts of a union type. Many other constructs can be expressed using case expressions. For example, conditional expressions are a syntactic shorthand for case expressions on booleans, *i.e.*,

```
if b then e1 else e2
```

is equivalent to

```
{case b of
  true  = e1
| false = e2}
```

[4]Note that union types also subsume enumerated types in Pascal.

Similarly, the clauses in a function definition are a syntactic shorthand for a case expression. For example, we could have written the carbons function as follows:

```
def carbons r =
    {case r of
      H            = 0
    | (C r1 r2 r3) = 1 + (carbons r1) + (carbons r2) + (carbons r3)} ;
```

Here is a useful help-function that uses ordering to check if one radical is less than or equal to another:

```
def le? r1 r2 = {  o = (ordering r1 r2)
                In
                  (o == lt) or (o == eq) } ;
```

3.1 More Efficient Representations

Unfortunately, the ordering computation described above is *very* inefficient! Consider supplying this function with two arguments that are equal, but deeply nested, radicals. The ordering function will traverse them both completely, calling carbons at every level. But carbons itself traverses the entire sub-structure. Thus, we will compute the carbons of an inner radical repeatedly, once for each carbon above it!

It seems unnecessary to construct a radical first, and then to traverse it again to compute its the number of carbons. As we build radicals, we can simultaneously count the carbons and "cache" it at each level in the radical structure itself. To do this, we change our type definition for radicals:

```
type radical = H | CI int radical radical radical ;
```

Here, the non-hydrogen structure is augmented with an integer component in which size of the radical will be cached. The carbons function is now trivial:

```
def carbons H                  = 0
  | carbons (CI nc r1 r2 r3) = nc ;
```

To construct a radical given its three components, we use the following function:

```
def C r1 r2 r3 = { nc1 = carbons r1 ;
                   nc2 = carbons r2 ;
                   nc3 = carbons r3
                 In
                   CI (1+nc1+nc2+nc3) r1 r2 r3 } ;
```

Finally, we can rewrite our ordering function (it is identical to the earlier version except that we do not have to call carbons to compute the number of carbons in a radical):

```
def ordering  H                  H                  = eq
  | ordering  H                  (CI nc2 r21 r22 r23) = lt
  | ordering (CI nc1 r11 r12 r13) H                  = gt
  | ordering (CI nc1 r11 r12 r13) (CI nc2 r21 r22 r23) =
             if      nc1 < nc2 then lt
             else if nc1 > nc2 then gt
else {case (ordering r11 r21) of
        lt = lt
      | gt = gt
      | eq = {case (ordering r12 r22) of
                 lt = lt
               | gt = gt
               | eq = (ordering r13 r23) }} ;
```

4 Generating Radicals

The first step in our program is to generate all radicals with `n` carbons, which we do inductively, following the definition of radicals. Assume that we have generated all radicals with less than `n` carbons. Then, for each triple `r1`, `r2` and `r3` such that `r1` $\leq$ `r2` $\leq$ `r3` and `(carbons r1) + (carbons r2) + (carbons r3)` $=$ `(n-1)`, we construct the radical `(C r1 r2 r3)` with `n` carbons.

We will first develop a transparent but slightly inefficient version of the radical generator in order to familiarize the reader with basic Id constructs. Later, we will analyze this solution to identify the inefficiencies and produce an efficient solution.

4.0.1 List and Array Comprehensions

Let us first concentrate on the sub-problem of generating a list containing all canonical 3-partitions `nc1, nc2, nc3` of a number `m`, *i.e.*, `nc1` $\leq$ `nc2` $\leq$ `nc3` and `nc1` $+$ `nc2` $+$ `nc3` $=$ `m`. One solution would be to write it using three nested loops, specified informally below:

```
"For each nc1 in the range 0 through m,
    For each nc2 in the range 0 through m,
        For each nc3 in the range 0 through m,
            Such that nc1 ≤ nc2 ≤ nc3
            and nc1 + nc2 + nc3 = m,
                Let the 3-tuple (nc1,nc2,nc3) be in the result list."
```

Rather than using loops, we use a more elegant notation called a *list comprehension*:[5]

```
def 3_partitions m =
       {: (nc1,nc2,nc3) || nc1 <- 0 to m
                         & nc2 <- 0 to m
                         & nc3 <- 0 to m when nc1 <= nc2
                                          and nc2 <= nc3
                                          and nc1+nc2+nc3 == m } ;
```

A list comprehension has the general form `{: eBody || generator & ... & generator }`. The generators nest from left to right, like loops. The expression `eBody` is evaluated inside the innermost loop and all the results are collected into a result list. A generator of the form `x <- e` repeatedly binds the identifier `x` to each element of the list `e`. Here, the expression "`0 to m`" evaluates to a list of integers from `0` through `m`. A filter "`when p`" can be attached to any generator, and has the effect of rejecting those combinations of variables that do not satisfy the predicate expression `p`. It is also

[5]Popularized by David Turner in his language KRC [26].

possible to have a generator of the form "x = e", which merely has the effect of binding identifier x to the value of e.

A closely related construct in Id is the *array comprehension*. For example, suppose we wished to compute an array containing the first n Fibonacci numbers ($n \geq 2$). Here is the definition:

```
fib_array = {array (1,n)
            | [1] = 1
            | [2] = 1
            | [j] = fib_array[j-1] + fib_array[j-2] || j <- 3 to n} ;
```

The prefix "{array (1,n)" introduces the array construct, and declares the array bounds to go from 1 through n. In Id, array bounds are determined at run time but, once allocated, arrays do not shrink or grow. The phrase "[1] = 1" indicates that, at index 1, the value is 1. Similarly, at index 2, the value is 1. Finally, at index j, for j ranging from 3 to n, the value is the sum of the previous two components. Each component of the array must be defined exactly once— a run-time error is detected if the clauses overlap, *i.e.*, if two different clauses attempt to define the same element j.

The ability to define both functions and data structures recursively is a fundamental part of Id. We say that data-structures are *non-strict*, *i.e.*, some components may be read while other components are as yet undefined. Such recursive data-structure definitions are also possible in non-strict functional languages such as Miranda [25] and Lazy ML [6], but not in strict functional languages such as SISAL [13], Scheme[20] and SML[14].[6] Recursive definitions of data-structures are notoriously difficult to parallelize. Id's write-once semantics for data structures, along with the hardware support in dataflow machines in the form of "I-structure" storage allow the programmer to use such definitions freely. We discuss this issue in Section 7.1.

The generators to the right of the "||" separator in list and array comprehensions have exactly the same syntax and semantics. Thus, list and array comprehensions are simply two different ways of packaging the results of the inner loop. Lists are used to create arbitrary length collections with subsequent linear-time access, whereas arrays are used to create known-length collections with constant-time access.

Let us try to improve our solution to the canonical partitions problem. We were generating m^3 triples only to discard most of them using the when clause (actually, more than $\frac{5}{6}$ of them are discarded). A little algebra makes it obvious that under the conditions imposed, nc1 cannot be more than $\frac{m}{3}$, nc2 cannot be more than $\frac{m-nc1}{2}$, and that nc3 = m - (nc1+nc2). Here is a more efficient solution that directly generates the required numbers:

```
def 3_partitions m =
       {: (nc1,nc2,nc3) || nc1 <- 0 to floor (m/3)
                         & nc2 <- nc1 to floor ((m-nc1)/2)
                         & nc3  =  m - (nc1 + nc2) } ;
```

Since the divisions may produce real-numbered results, we use the function floor to truncate them back to integers.

[6]Miranda and Lazy ML achieve non-strictness *via* a technique known as *lazy evaluation*, which permits the manipulation of "infinite" data structures but also introduces some inefficiency in implementation. Lazy evaluation is entirely unnecessary in this problem, since we are dealing only with finite data structures.

4.1 First Attempt

Let us now look at the inductive step in generating radicals of size n. Let us first assume that we are given an array radicals such that at each index j < n, radicals[j] contains the ordered list of radicals with j carbons. This is depicted pictorially in Figure 1, where the ⊥ symbols indicate that the array components are as yet undefined beyond index 3.

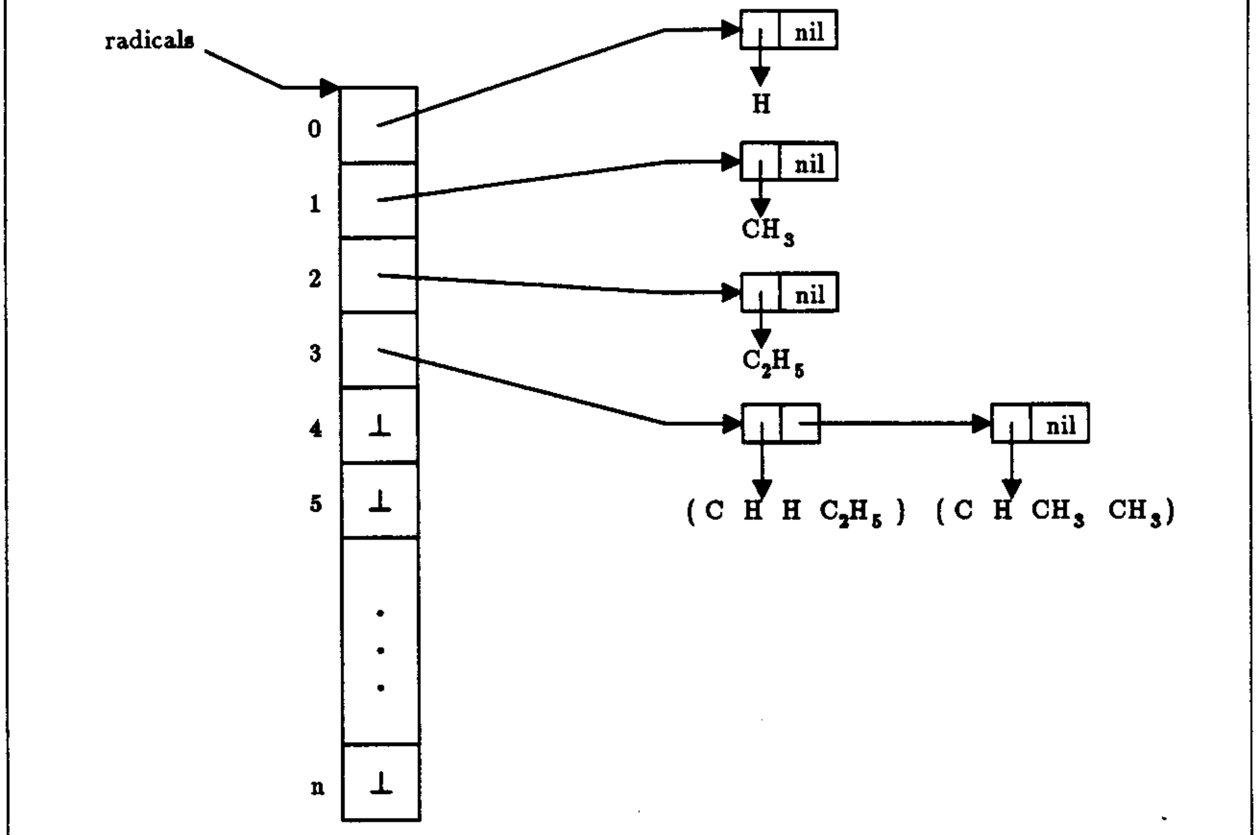


Figure 1: The **radicals** array

Here is a function for the inductive step. It generates the ordered list of radicals with n carbons, assuming it is given radicals with less than n carbons:

```
def rads_of_size_n radicals n =
    {: C r1 r2 r3 || (nc1,nc2,nc3) <- 3_partitions (n-1)
                   & r1 <- radicals[nc1]
                   & r2 <- radicals[nc2] when (le? r1 r2)
                   & r3 <- radicals[nc3] when (le? r2 r3) } ;
```

The first generator produces the canonical 3-partitions of radical sizes totalling (n-1). The second generator produces all radicals r1 with nc1 carbons. The third and fourth generators produce all radicals r2 and r3 with nc2 and nc3 carbons, respectively, and we ensure canonical ordering using the when clauses. Finally, we collect together all radicals of the form (C r1 r2 r3) into the result list.

All that remains is to package the base step and the inductive step to produce all radicals of size up to n. Here is a definition of the array radicals with indices 0 through n, such that the j'th component is an ordered list of all radicals with j carbons:

```
radicals = {array (0,n)
           | [0] = H:nil
           | [j] = rads_of_size_n radicals j || j <- 1 to n}
```

At index 0 (representing radicals with 0 carbons), the array contains a singleton list containing the radical H, expressed using the notation H:nil. Here, the constant nil is the empty list, and ":" is an infix "cons" operator that a adjoins an element to a list. At each index j in the range 1 through n, the array contains the list computed by (rads_of_size_n radicals j).

Note that the definition for the array radicals is recursive, *i.e.*, the array being defined is itself passed as an argument to rads_of_size_n within the array definition. This is a reflection of the induction, or recurrence relation with which radicals are defined, and relies on Id's non-strict evaluation of data structures.

4.2 Critique of rads_of_size_n

Let us concentrate on that fragment of rads_of_size_n that produces radicals r1 and r2 in canonical order (with nc1 and nc2 carbons, respectively). When nc1 < nc2, we know that r1 < r2, by definition. It is only at the boundary condition when nc1 = nc2 that r1 and r2 may be in the wrong order. To protect against this, we used the filter "when (le? r1 r2)". The inefficiency is this: we are unnecessarily performing radical comparisons even when nc1 < nc2.

We can fix this immediately as follows:

```
def rads_of_size_n radicals n =
    {: C r1 r2 r3 || (nc1,nc2,nc3) <- 3_partitions (n-1)
                   & r1 <- radicals[nc1]
                   & r2 <- radicals[nc2] when (if (nc1 == nc2) then (le? r1 r2)
                                               else true)
                   & r3 <- radicals[nc3] when (if (nc2 == nc3) then (le? r2 r3)
                                               else true) } ;
```

so that the radical-comparison is done only when nc1 = nc2. This may be worthwhile because the equality test on the numbers nc1 and nc2 is likely to be cheaper than the radical comparison.

Another possible fix is this:

```
def rads_of_size_n radicals n =
    {: C r1 r2 r3 || (nc1,nc2,nc3) <- 3_partitions (n-1)
                   & r1 <- radicals[nc1]
                   & r2 <- if (nc1 == nc2) then
                             {: r2temp || r2temp <- radicals[nc2] when (le? r1 r2temp)}
                           else
                             radicals[nc2]
                   & r3 <- if (nc2 == nc3) then
                             {: r3 || r3 <- radicals[nc3] when (le? r2 r3)}
                           else
                             radicals[nc3] } ;
```

which brings the equality test outside, *i.e.*, when nc1 ≠ nc2, we directly use the list radicals[nc2]; otherwise we filter the list. But in this solution we generate many extra intermediate lists which may be difficult to optimize away.

In any case, there is still a lingering inefficiency. Even if the filtering is limited to the case when `nc1` = `nc2`, we are still generating many `r1`, `r2` pairs that are out of order, only to have them filtered out subsequently. For example, let `nc1` = `nc2` = `3`. `R1` and `r2` will be drawn from `radicals[3]`. Referring back to Figure 1, each will be bound, in turn, to the values `(C H H H` C_2H_5`)` and `(C H H` CH_3 CH_3`)`. But when `r1` is `(C H H` CH_3 CH_3`)` and `r2` is `(C H H H` C_2H_5`)`, they are out of order. In general, we are generating n^2 pairs and rejecting half of them.

4.3 Second Attempt

The basic problem in `rads_of_size_n` is this: when picking a radical `r1` from the list `radicals[nc1]`, we do not keep track of the position in the list that `r1` was drawn from. Thus, when we pick radicals `r2` from the same list (*i.e.*, when `nc1` = `nc2`), we pick some radicals from earlier positions in the list, so that `r1` and `r2` are not in canonical order.

To correct this, whenever we pick an `r1` from the list `radicals[nc1]`, we will also keep the *remainder* of the list starting at that radical, so that whenever `nc1` = `nc2`, we will pick `r2`'s from this remainder instead of the full list `radicals[nc2]`. In this way, we never pick any `r2`'s that are earlier than `r1`.

Similarly, whenever we pick an `r2` from a list, we keep with it the remainder of the list starting at that radical, so that whenever we wish pick `r3`'s with the same size, we pick it from this remainder instead of the full list `radicals[nc3]`.

For this purpose, we first define a help-function called `remainders` that takes a list of radicals:

$r_a, r_b, r_c \cdots$

and produces a list of remainders of this list, *i.e.*, a list of lists of radicals:

$(r_a, r_b, r_c, r_d, r_e, \ldots), (r_b, r_c, r_d, r_e \ldots), (r_c, r_d, r_e \ldots), \ldots$

Then, we can pick `r1`'s from the heads of these lists, to get $r_a, r_b, r_c, \ldots$, but for each of them, we also have available the remainder of the list, from which we can pick `r2`'s, ensuring canonical order. For example, when we pick `r1` to be r_c, the first element of the third list, we can also pick `r2`'s from the third list, to get $r_c, r_d, r_e, \ldots$. Here is the code to generate the remainders list:

```
def remainders nil     = nil
  | remainders (r:rs) = (r:rs) : (remainders rs) ;
```

Note that `remainders` is actually a general-purpose function— it will work on lists of anything, not just radicals.

Here is the efficient version of `rads_of_size_n`:

```
def rads_of_size_n radicals n =
    {: C r1 r2 r3 || (nc1,nc2,nc3) <- 3_partitions (n-1)
                   & r1:r1s <- remainders (radicals[nc1])
                   & r2:r2s <- remainders (if (nc1 == nc2) then r1:r1s
                                           else radicals[nc2])
                   & r3     <- if (nc2 == nc3) then r2:r2s
                               else radicals[nc3] } ;
```

Note that in this version, we do no radical comparisons at all, *i.e.*, the functions `le?` and `ordering`, which were useful for expository purposes, are not really necessary.

Finally, we abstract our definition of the array `radicals` into a function on `n`:

```
def radical_generator n =
    { radicals = {array (0,n)
                | [0] = H:nil
                | [j] = rads_of_size_n radicals j || j <- 1 to n}
    In
      radicals} ;
```

5 Paraffins from Radicals

5.1 Representation and Canonical Forms for Paraffins

Suppose we chose the following simple representation for paraffins:

```
type paraffin = ROOT_CARBON radical radical radical radical ;
```

i.e., a root carbon connected to four radicals. To avoid duplicates, we may impose the condition that the radicals in a paraffin `ROOT_CARBON r1 r2 r3 r4` must be ordered, *i.e.*, $r1 \leq r2 \leq r3 \leq r4$. Unfortunately, this is not sufficient to guarantee uniqueness of representation, as shown by the following example:

1. Attach radicals H, H, CH_3, and CH_3 to a carbon atom to get:

   ```
   ROOT_CARBON H H (C H H H) (C H H H)
   ```

2. Attach radicals H, H, H, and C_2H_5 to a carbon atom to get:

   ```
   ROOT_CARBON H H H (C H H H (C H H H))
   ```

Despite satisfying the ordering constraint, both represent the same molecule.

The solution is straightforward— rather than picking an arbitrary root in a paraffin, we define a unique "center" such that it is "balanced" on all sides.[7] We define the *center* of a paraffin with n carbons to be:

- either a *bond* with radicals of size $\frac{n}{2}$ on either side,
- or a carbon atom connected to four radicals such that all of them have size strictly less than $\frac{n}{2}$.

In the former case, we call it a *bond-centered* paraffin (BCP), and in the latter, a *carbon-centered* paraffin (CCP). We leave it as an exercise to the reader to prove that the definition is sound, *i.e.*, that the center is uniquely defined, and that every paraffin is either a BCP or a CCP, but not both.

We can express this in the representation of paraffins:

```
type paraffin = BCP radical radical | CCP radical radical radical radical ;
```

Having defined unique centers, we can avoid duplicates as follows:

- In a bond-centered paraffin (`BCP r1 r2`), we require that $r1 \leq r2$.
- In a carbon-centered paraffin (`CCP r1 r2 r3 r4`), we require that $r1 \leq r2 \leq r3 \leq r4$.

[7]That is, it is straightforward in retrospect. One of the authors (Heller) discovered several possible definitions for the "center" of a paraffin, based on carbon counts, size of longest carbon chains, *etc.* We were also assisted in this direction by hints in some code for this problem that we received from Olaf Lubeck and Vance Faber of Los Alamos National Laboratories.

Finally, (sigh! need we be surprised?) we discovered that this problem has been discussed extensively by Knuth [11]. It turns out that it is identical to the problem of finding a unique center for what Knuth calls *free trees*, *i.e.*, trees for which the root is unknown.

5.2 Generating Bond-Centered Paraffins

Here is a function to generate all bond-centered paraffins with j carbons, given that we have already computed the array radicals where radicals[i] contains the list of canonical radicals with i carbons in canonical order:

```
def BCP_generator radicals j =
        if odd? j then nil
        else
          {: BCP r1 r2 || r1:r1s <- remainders (radicals[floor (j/2)])
                      & r2 <- r1:r1s } ;
```

From the definition, we can see that the size of a bond-centered paraffin must be even, so that if j is odd, the list of BCPs is empty. When j is even, the radicals on either side of the bond must have $\frac{j}{2}$ carbons. We need pairs of radicals r1 and r2 with this carbon count, but to avoid duplicates, we need them in canonical order. So, we use the same trick as before— when picking an r1 from the list, we also keep with it the remainder of the list, so that we can pick r2 from that remainder, thus guaranteeing canonical order.

5.3 Generating Carbon-Centered Paraffins

As before, let us first concentrate on the problem of generating all canonical 4-partitions nc1, nc2, nc3, nc4 of a number m, representing the sizes of the four radicals of a carbon-centered paraffin of size m+1. We know that:

1. nc1,nc2,nc3,nc4 $< \frac{m}{2}$, by the definition of "center".
3. nc1 $+$ nc2 $+$ nc3 $+$ nc4 $=$ m, and
2. nc1 $\leq$ nc2 $\leq$ nc3 $\leq$ nc4, by our canonical ordering.

Under these conditions, it is obvious that:

- nc1 ranges from 0 to $\frac{m}{4}$;
- nc2 ranges from nc1 to $\frac{m-nc1}{3}$;
- Normally, nc3 ranges from nc2 up to $\frac{m-nc1-nc2}{2}$, and
- nc4 = M - nc1 - nc2 - nc3.

It is clear that nc1, nc2 and nc3 satisfy condition 1, above. Unfortunately, when they are all small, nc4 can become greater than $\frac{m}{2}$. We can avoid generating large nc4's by imposing an additional condition on nc3 so that it is always large enough to keep nc4 $< \frac{m}{2}$. We ensure that nc3 must be at least $\frac{m}{2}$-nc1-nc2. Here is the code to generate the partitions:

```
def 4_partitions m =
        {: (nc1,nc2,nc3,nc4) || nc1 <- 0  to floor (m/4)
                              & nc2 <- nc1 to floor ((m-nc1)/3)
                              & nc3 <-    (max nc2 (ceiling (m/2-nc1-nc2)))
                                       to (floor ((m-nc1-nc2)/2))
                              & nc4 =  m - nc1 - nc2 - nc3 } ;
```

The ceiling function converts a fraction to the next integer.

Assume, again, that we are given the array radicals where radicals[i] contains the list of canonical radicals with i carbons in canonical order. Here is the function to generate all carbon-centered paraffins with j carbons:

```
def CCP_generator radicals j =
        {: CCP r1 r2 r3 r4 || (nc1,nc2,nc3,nc4) <- 4_partitions (j-1)
                              & r1:r1s <- remainders (radicals[nc1])
                              & r2:r2s <- remainders (if nc1==nc2 then r1:r1s
                                                      else radicals[nc2])
                              & r3:r3s <- remainders (if nc2==nc3 then r2:r2s
                                                      else radicals[nc3])
                              & r4 <- if (nc3==nc4) then r3:r3s
                                      else radicals[nc4] } ;
```

5.4 The solution to the Paraffins Problem

Here, finally, is the solution to the paraffins problem:

```
def paraffins_until n =
    {
      radicals = radical_generator (floor (n/2)) ;
    In
      {array (1,n)
      | [j] = (BCP_generator radicals j),
              (CCP_generator radicals j)    || j <- 1 to n}} ;
```

Because our paraffins are "centered", we need generate radicals only up to size $\frac{n}{2}$. The result is an array of pairs of lists. The j'th index of the array contains all paraffins with j carbons, represented as a pair— a list of bond-centered paraffins and a list of carbon-centered paraffins with j carbons.

6 Parallelism

In our development of the solution, we did not pay any explicit attention to parallelism at all, concentrating, instead, on correctness and clarity of expression. Let us now turn our attention to the parallelism issue. Here are some interesting statistics concerning the size of the problem:

n	Number of paraffins with n carbons	Number of radicals with n carbons	n	Number of paraffins with n carbons	Number of radicals with n carbons
0	—	1	9	35	211
1	1	1	10	75	507
2	1	1	11	159	1238
3	1	2	12	355	3057
4	2	4	13	802	7639
5	3	8	14	1858	19241
6	5	17	15	4347	48865
7	9	39	16	10359	124906
8	18	89	17	24894	321198

There are some opportunities for parallelism that are fairly apparent. For example, it is clear that the computation of paraffins of size i can proceed in parallel with the computation of paraffins of size j, where i $\neq$ j. Also, for a given size j, the computation of bond-centered paraffins can proceed in parallel with the computation of carbon-centered paraffins. These would be fairly easy to express in a language with annotations for parallelism.

There is also much producer-consumer parallelism in the program. For example, the computation of the list `radicals[j]` can begin soon after the initial segments of the lists `radicals[i]`, where `i < j`, have been computed. The computation of `paraffins[k]` can begin soon after the initial segments of the lists `radicals[i]`, where `i` < $\frac{k}{2}$, have been computed. There is a similar kind of overlap between `radical_generator` and `3_partitions`, and between `CCP_generator` and `4_partitions`. This kind of parallelism is difficult to express using explicit annotations.

In Id, *all* this parallelism is available *automatically* to the programmer. Rather than abandoning the pursuit of parallelism at the iteration or procedure level, the Id compiler goes all the way down to the level of individual instructions. It is possible to do this *a)* because of the single-assignment (functional) semantics of Id, and *b)* because the synchronization of the producer-consumer is assumed to be done in hardware (as in a dataflow machine), so that the compiler does not have to worry about it. A program is translated into a *dataflow graph*, which is a parallel machine language in which each instruction directly designates its successors (there can be more than one)[8] An instruction may also have multiple predecessors, so that a dataflow graph describes a *partial order* on instructions. The partial order specifies *only* data dependencies, *i.e.,* instruction j is a successor of instruction i if and only if the value produced by i is used by j. Because of the lack of anti-dependencies, aliasing, *etc.* [17], the translation of Id programs to dataflow graphs is relatively straightforward.

We can execute dataflow graphs on GITA, our dataflow graph interpreter [16]. In the first step, we execute the initial instructions in the dataflow graph (*i.e.,* the "roots" of the partial order of instructions). In each subsequent step, we execute all instructions for which all their predecessors have executed, and we repeat this until the program terminates. Notice that there is no separate "program counter" that schedules instructions. Finally, we can plot the "parallelism profile" of the run, which measures how many instructions were executed at each time step. Because Id and the compiler do not introduce any dependencies that were not originally in the algorithm, the parallelism profile is, in a sense, a measure of the *inherent* parallelism of the algorithm, an "ideal" reference point against which to measure how much parallelism is actually exploited by a particular machine.

The parallelism profile for generated by GITA for `(paraffins_until 15)` is shown in Figure 2. At the point of maximum parallelism, a little over 1700 instructions were executed concurrently. The total instruction count was 533,503, and the critical path length (number of time steps in which the program completed) was 4857 instructions. A detailed interpretation of parallelism profiles is beyond the scope of this paper; we refer the reader to [2].

7 Conclusion

In this paper we have made the point that for future, general-purpose parallel programming, we can kill two birds with one stone by moving to languages such as Id. We can not only raise the level of programming significantly and guarantee determinacy, but we can also generate code with abundant parallelism, which is necessary if we are to exploit parallel hardware effectively. Higher-order functions, non-strictness, list and array comprehensions all play a major role in this, as does the declarative nature of the language. We will elaborate on these points in this section.

[8]Which is unlike sequential machine languages, where each instruction has a single successor, and, except in jumps, the successor is the instruction that follows textually.

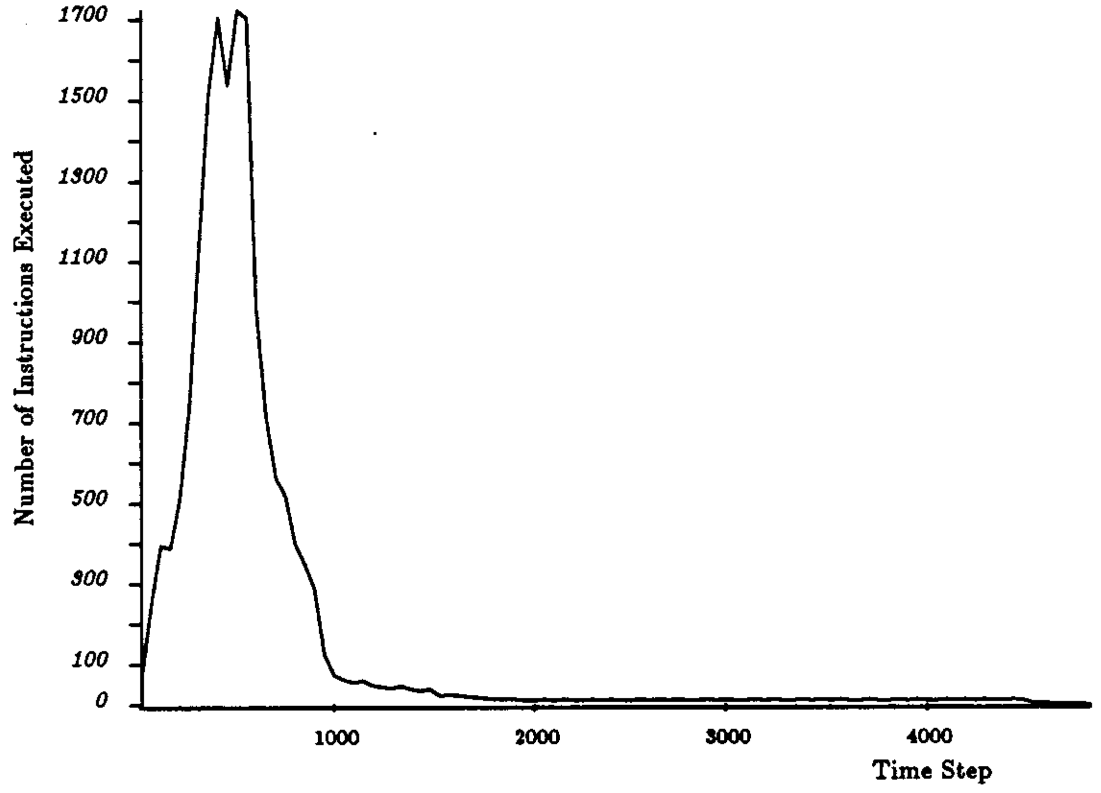


Figure 2: Parallelism profile for (`paraffins_until 15`)

7.1 Level of Coding: Programming without List and Array Comprehensions

List and array comprehensions constitute a powerful, compact and intuitive notation.[9] As an illustration, consider the problem of producing the squares of all integers from 1 to n. Using list comprehensions, we write:

```
{: sqr i || i <- 1 to n}
```

To write it without list comprehensions, we may use the function **map** given below:

```
def map f nil    = nil
 |  map f (x:xs) = (f x):(map f xs) ;
```

Map takes a function and a list as arguments, applies the function to each member of the list, and returns the list of results. The solution to our problem, then, is:

```
map sqr (1 to n)
```

Suppose we wanted the squares of only the *even* integers from 1 to n. Again, using list comprehensions, we write:

```
{: sqr i || i <- 1 to n when even? i}
```

To write it without list comprehensions, we need a filter:

```
def filter p nil    = nil
 |  filter p (x:xs) = if (p x) then x:(filter p xs)
                      else filter p xs            ;
```

Filter takes a predicate function and a list as arguments, and returns a list containing only those elements that satisfy the predicate. The solution to our problem, then, is:

```
map sqr (filter even? (1 to n))
```

The solution is not very efficient, because **filter** produces an intermediate list that is immediately discarded after it has been consumed by **map**.

Now let us complicate the problem somewhat. Consider the problem of generating the indices of an $n \times n$ matrix, in the order (1,1), (1,2), ..., (1,n), (2,1), (2,2), ..., (n,n). In other words, we are generating the cross-product of two lists going from 1 to n. Using list comprehensions, it is easy:

```
{: pair i j || i <- 1 to n & j <- 1 to n}
```

(Here, we have written "`(pair i j)`" instead of the semantically identical "`(i,j)`" in order to simplify subsequent examples.) Here is a solution that does not use list comprehensions:

```
{ def f nil    js = nil
   |  f (i:is) js = (map (pair i) js) ++ (f is js)
In
  f (1 to n) (1 to n) }
```

[9]The reader may be interested in pursuing the observation that list comprehensions subsume the SELECT ... FROM ... WHERE ... construct in the popular relational database query language SQL[1].

Here, f is applied to the two lists going from 1 to n representing the i's and the j's. It recursively iterates over i's; at each i, it maps (pair i) over all the j's. This returns a list (i,1), (i,2), ..., (i,n). Finally, all these lists are *appended* together (using the built-in append operator "++") to produce the final list.

An inefficiency in this solution is due to the append operation. In the expression (e1 ++ e2), each sub-expression produces an intermediate list, and these are immediately discarded after the append operator consumes them. Further, the append involves a linear-cost traversal of the list produced by e1, and this is done repeatedly, once for each i.

Let us complicate the problem further. Suppose we wished to generate the indices of only the right-upper triangle of the matrix, *i.e.*, (1,1), (1,2), ..., (1,n), (2,2), (2,3), ..., (2,n), (3,3), ..., (n,n). Again, using list comprehensions, it is easy:

```
{: pair i j || i <- 1 to n & j <- i to n}
```

The solution without list comprehensions, above, cannot be changed so easily— we cannot simply replace the second (1 to n) by (i to n). The problem is that here, we need to generate a *dependent* or *relative* cross product, *i.e.*, the list of j's is defined relative to each i. We can fix the solution as follows:

```
{ def f nil    = nil
  | f (i:is) = (map (pair i) (i to n)) ++ (f is)
In
  f (1 to n) }
```

If we wanted to impose a further condition on the desired indices, such as "for only even values of i+j", we'd have to introduce a filter:

```
{ def p i j = even? (i+j) ;
  def f nil    = nil
   | f (i:is) = (map (pair i) (filter (p i) (i to n))) ++ (f is)
In
  f (1 to n) }
```

The list comprehension solution:

```
{: pair i j || i <- 1 to n & j <- i to n when even? (i+j)}
```

As the number of nested generators and predicates increases, it becomes difficult to continue this *ad hoc* process of developing a solution using recursive functions. The reader is referred to [18], Chapter 7, where a systematic, general method is given to translate list comprehensions. The method is ultimately powerful enough to eliminate all the intermediate list-building performed by map's, filter's and appends.[10] By the time all these optimizations are applied, the program can become extremely obscure compared to its counterpart using comprehensions.

[10]It is interesting to note that the optimization amounts to a data-structuring analog of the "continuation-passing transformation" used in Scheme compilers [23,12]. Rather than returning an intermediate list from map and then appending it the "rest-of-the-result-list", we send the rest-of-the-result-list in as an extra *argument* to map and leave it to map to attach it to the end of the list that it produces.

7.2 Loops and Open Lists

Even if we were to write the obscure solution to the above examples that was "optimal" in the sense that it produced no intermediate lists, we are still left with a basic inefficiency. The essence of the problem is this. In all architectures that we are aware of, it is always better if we can express something using a loop instead of general procedure calls. In many implementations of functional languages, the compiler automatically implements a certain category of recursive functions, called "tail-recursive" functions, using loops. Unfortunately, the functions `map`, `filter`, "++", *etc.* are not tail-recursive, and hence are not subject to this optimization.

Let us again look at the problem of generating all indices of a $n \times n$ matrix in the order specified above. Consider this solution using a (functional) loop, written in Id:

```
{ result = nil
In
  {for i <- 1 to n do
     next result = {for j <- 1 to n do
                        next result = (i,j) : result
                      finally result}
   finally result}}
```

In Id, a loop is an *expression*, *i.e.*, it evaluates to a value. The phrase "`next result = e`" binds the value of `e` to the identifier `result` for the *next* iteration of the loop. The phrase "`finally result`" returns, as the value of the loop, the value of `result` in the last iteration.

Clearly, this solution is optimal in that it directly creates the result list; no extra intermediate lists are created. The only problem is: the result list is in the wrong order ((n,n), (n,$n-1$), ..., (1,2), (1,1))! How do we create the list in the right order? We could reverse the direction of the loops, by saying "`for i <- n downto 1 do ...`", but this unacceptable in general because we may have `while` loops that cannot be reversed. We could change the list construction in the inner loop so that, instead of adjoining the new pair to the front of the result list, we appended it to the end:

```
...
     next result = result ++ ((i,j):nil)
...
```

Unfortunately, this is back to square one— it is *very* extravagant in space and time, as each append operation takes order n time and space! We could bite the bullet, produce the answer in reverse order, and write a subsequent loop to reverse it, but this still doubles the space and time.

The basic problem is this: there is no efficient way to "grow" the end of the result list in a purely functional setting. In Id, it is possible to solve this problem by stepping out of the purely functional subset and using a construct known as "I-structures" [5]. In functional programming languages, it is not possible to allocate an "empty" data structure and separately fill its slots— a data structure and its components are always defined simultaneously. I-structures allow this separation, and this is exactly what we need to solve this problem. At first glance, filling an I-structure slot will appear to be an ordinary assignment statement, but there is a crucial difference— I-structures slots are "write-once" locations; we will return to this issue at the end of this section.

We use a technique in Id known as "open lists".[11] A *cons cell* is a data structure with two slots (at indices 1 and 2, respectively). A list is a chain of cons cells; in each cons cell, slot 1 contains

[11] Similar to the technique of "difference lists" in logic programming.

an element of the list, and slot 2 contains a reference to the next cons cell in the chain. An open list is one in which slot 2 of the last cons cell is *empty*.

We define an *open list* by a pair of references, one to the first cons cell in the list and the other to the last cons cell in the list. An empty open list is simply a pair of references to the same cons cell. We first define some abstractions to manipulate open lists. Assume that we have a function new_cons_cell that allocates a new, empty cons cell. Here is a function that creates a new, empty, open list:

```
def make_empty_ol () = { ol = new_cons_cell ()
                       In
                         ol,ol } ;
```

i.e., is allocates a new cons cell and returns two references to it. Here is a predicate that tests for the empty open list (where eq? tests if its two argument references are identical):

```
def empty_ol? (first,last) = (eq? first last) ;
```

Here is a function that, given an open list and a value, extends the open list by "growing" the tail of the open list with a new cons cell containing this value, and returns the new open list:

```
def extend_ol (first,last) x = { new_last = new_cons_cell () ;
                                  new_last[1] = x ;
                                  last[2] = new_last
                                In
                                  first,new_last } ;
```

The second line in the block is an I-structure assignment that stores x in the first slot of the new cons cell; the next line is an I-structure assignment that connects the current last cell to the new last cell. Finally, we return references to the head and to the new last cell. Note that the first cons cell in an open list is "unused". It simply acts as a stub onto which we can attach new cons cells containing the real elements of the list.

Here is a function that, given two open lists, appends them together and returns the resulting open list:

```
def append_ol (first1,last1) (first2,last2) =
    if (empty_ol? (first2,last2)) then
      (first1,last1)
    else
      { last1[2] = first2[2]
      In
        first1, last2 } ;
```

If the second open list is empty, it simply returns the first open list as is; otherwise, it connects the end of the first open list to the second open list, and returns the references to the extremities of this appended list.

Finally, here is a function that, given an open list, "closes" the empty slot at the end using nil, the empty list, and returns a reference that can be used as a normal list.

```
def close_ol (first,last) =
    { last[2] = nil
    In
      first[2] } ;
```

We can now express our matrix index-generation problem in Id:

```
{ ol_i = make_open_list () ;
  final_ol_i = {for i <- 1 to n do
                   ol_j = make_open_list () ;
                   final_ol_j = {for j <- 1 to n do
                                   next ol_j = extend_ol ol_j (i,j)
                                 finally ol_j} ;
                   next ol_i = append_ol ol_i final_ol_j
                 finally ol_i}
In
  close final_ol_i}
```

The compiler for Id automatically translates list comprehensions into an open list form (with several optimizations that avoid unused cells at the heads of open lists, *etc.*).

A similar translation is, of course, also possible in any sequential language, such as FORTRAN or Lisp. In fact, a good compiler for a functional language may translate list comprehensions into von Neumann code of this form. However, a crucial issue is this: Id treats I-structure slots as "write-once" locations, and this has a major impact on compiling, determinacy and parallelism.

An I-structure location has extra "presence" bits that indicate whether a value is present or whether the location is empty. When an I-structure is allocated (*e.g.*, in `new_cons_cell`), all its slots are initialized to the empty state. Note that by "empty" we do not mean `nil`, the empty list; rather, "empty" literally means "no value yet" (for semanticists, "empty" is identical to $\perp$, or "no information"). When a consumer attempts to read an empty slot, it is automatically suspended until a producer writes a value there, at which time the consumer is automatically resumed. It is a runtime error to attempt to write a value into a non-empty location. Thus, Id's I-structure assignments are "safe", in that they do not compromise determinacy and, because they are synchronized automatically, they do not limit parallelism. The presumption of I-strucure semantics makes compilation significantly easier, as it obviates the concern for introducing explicit synchronization. I-structure semantics are not easy to simulate in software. The Id compiler presumes hardware support for presence bits and the automatic suspension and resumption of processes. Such support is fundamental to dataflow architectures [4].

Suppose there is one function that produces an open list and another that consumes it, and we wish to overlap their execution. If the consumer overtakes the producer, it will attept to read a value from the empty slot at the end of the open list. With Id's I-structure semantics, the consumer will automatically suspend until the producer has extended the list further.

7.3 Comparison with Other Functional Languages

Many of the language features of Id that we have presented here are also available, to varying degrees, in other functional programming languages.

ML [14] and Scheme [20], while containing some imperative features, have subsets that are pure, higher-order functional languages. Common Lisp [22], too, can be used in this way, though its distinction between function values and other values complicates higher-order programming. These languages have *strict* semantics which, while simplifying compilation, also reduce their expressive power in not being able to define data structures with recurrences. Strict semantics, if retained in parallel versions, would also limit producer-consumer parallelism.

Miranda [25] (and its predecessors SASL [24] and KRC [26]) and LML [6] are purely functional languages with non-strict semantics, which is achieved *via* lazy evaluation. The choice of lazy evaluation gives the programmer a very powerful tool— the ability to use "infinite" data structures freely. Unfortunately, lazy evaluation also induces a significant performance penalty even on those programs that do not need it.

Lucid is another "dataflow" language [27], though the dataflow graphs used to interpret Lucid programs are at a much too abstract level to be considered a machine language. Lucid omits higher-order functions, but like Miranda and LML, uses infinite data structures and relies on lazy evaluation.

To date, compilation of the languages cited above has focused on sequential machines.

SISAL is a parallel functional language [13]. It deliberately omits higher-order functions in favor of simplicity, and has strict semantics, though a special construct called "streams" can be used explicitly for producer-consumer parallelism. SISAL programs are compiled to IF1, an intermediate language of dataflow graphs [21]. Most current SISAL research focuses on compiling to existing multiprocessors, except at Manchester, where the target is the Manchester dataflow machine.

Amongst all these languages, Id is the only one that we are aware of that gives an efficient treatment of arrays.

This variety is a sign of the vitality of the field— we have barely scratched the surface in the study of functional languages and their compilation. There is currently an effort underway to produce a common functional language called Haskell to facilitate exchange between researchers in the field [28].

Researchers at Yale [8,7]) and University College London [19] are actively investigating the parallel implementation of higher-order, non-strict functional languages. Their preference is for lazy evaluation, which, as we have pointed out before, can induce a performance penalty even on programs that do not require it. The compiler uses information from *strictness analysis* and, perhaps, programmer annotations, to alleviate this. Programs are partitioned explicitly by the compiler into parallel tasks, each of which is compiled to conventional von Neumann code. A major issue is deciding how to do this partitioning [9].

In contrast, Id has evolved, from the beginning, with implicit, fine-grained parallelism in mind [3]. Each feature in Id was included only after it was understood how to compile it to dataflow graphs. The compilation of Id continues to be a major research effort. In this, we are assisted by the abundant parallelism arising out of Id's declarative nature. Our target is to be competitive with the *fastest* languages available on parallel supercomputers. We believe that, simultaneously, Id will also raise the level of programming.

References

[1] *Database Language SQL.* American National Standards Institute, 1986. (also International Standards Organization Document ISO/TC97/SC21/WG3 N117.

[2] Arvind, D. E. Culler, and G. K. Maa. *Assessing the Benefits of Fine-grained Parallelism in Dataflow Programs.* Technical Report CSG Memo 279, Computation Structures Group, MIT Lab. for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 1988.

[3] Arvind, K. P. Gostelow, and W. Plouffe. *The (preliminary) Id report.* Technical Report 114, Dept.of Information and Computer Science, University of California, Irvine, CA, 1978.

[4] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. In *Proceedings of the PARLE Conference, Eindhoven, The Netherlands. (LNCS Volume 259)*, Springer-Verlag, June 15-19 1987.

[5] Arvind, R. S. Nikhil, and K. K. Pingali. I-Structures: Data Structures for Parallel Computing. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, (Springer-Verlag LNCS 279).*, pages 336–369, September/October 1986.

[6] L. Augustsson and T. Johnsson. *Lazy ML User's Manual.* Technical Report (Preliminary Draft), Programming Methodology Group Report, Department of Computer Science, Chalmers University of Technology and University of Goteborg, S-421 96 Goteborg, Sweden, January 1988.

[7] B. Goldberg and P. Hudak. Alfalfa: Distributed Graph Reduction on a Hypercube Multiprocessor. In *Proceedings of the Workshop on Graph Reduction, Santa Fe, New Mexico, USA, (Springer-Verlag LNCS 279).*, pages 94–113, September/October 1986.

[8] P. Hudak. *ALFL Reference Manual and Programmer's Guide.* Technical Report YALEU/ DCS/ RR-322, Second Edition, Department of Computer Science, Yale University, October 1984.

[9] P. Hudak and B. Goldberg. Serial Combinators: "Optimal" Grains of Parallelism. In *Proceedings of Functional Programming Languages and Architectures, Nancy, France*, pages 382–399, September 1985. Lecture Notes in Computer Science 201, Springer-Verlag, Berlin.

[10] A. H. Karp. Programming for Parallelism. *IEEE Computer*, 43–57, May 1987.

[11] D. E. Knuth. *The Art of Computer Programming, Volume 1/ Fundamental Algorithms.* Addison Wesley, Reading, Massachusetts, 1973.

[12] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An Optimizing Compiler for Scheme. *ACM SIGPLAN Notices*, 21(7):219–233, July 1986. (Proceedings of the SIGPLAN 86 Symposium on Compiler Construction).

[13] J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, P. Hohensee, and I. Dobes. *SISAL Reference Manual.* Technical Report, Lawrence Livermore National Laboratory, 1984.

[14] R. Milner. A Proposal for Standard ML. In *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 184–197, August 1984.

[15] R. S. Nikhil. *Id (Version 88.0) Reference Manual.* Technical Report CSG Memo 284, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, March 25 1988.

[16] R. S. Nikhil. *Id World Reference Manual.* Technical Report, Computation Structures Group, MIT Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, April 1987.

[17] D. A. Padua and M. J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12), December 1986.

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

[19] S. L. Peyton-Jones, C. Clack, J. Salkild, and M. Hardie. GRIP – A High Performance Architecture for Parallel Graph Reduction. In *Proceedings of the 3rd. International Conference on Functional Programming and Computer Architecture, Portland, Oregon*, September 1987.

[20] J. Rees and W. Clinger. *Revised^3 Report on the Algorithmic Language Scheme.* Technical Report, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, 1986.

[21] S. Skedzielewski and J. Glauert. *IF1: An Intermediate Form for Applicative Languages.* Technical Report M170, Lawrence Livermore National Laboratory, Livermore, California, July 31 1985.

[22] G. L. Steele Jr. *Common Lisp: The Language.* Digital Press, Billerica, Massachusetts, 1984.

[23] G. L. Steele Jr. *RABBIT: A Compiler for SCHEME.* Technical Report AI-TR-474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, Cambridge, MA, May 1978.

[24] D. A. Turner. A New Implementation Technique for Applicative Languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[25] D. A. Turner. Miranda: A Non-Strict Functional Language with Polymorphic Types. In *Proc. Functional Programming Languages and Computer Architecture, Nancy, France (Springer-Verlag LNCS 201)*, pages 1–16, September 1985.

[26] D. A. Turner. The Semantic Elegance of Applicative Languages. In *Proc. ACM Conference on Functional Programming Languages and Computer Architecture, Portsmouth, New Hampshire*, pages 85–92, October 1981.

[27] W. W. Wadge and E. A. Ashcroft. *Lucid, The Dataflow Programming Language.* Academic Press, London, 1985.

[28] P. Wadler (*editor*), Arvind, B. Boutel, J. Fairbairn, J. Fasel, P. Hudak, J. Hughes, T. Johnsson, R. Kieburtz, J. Launchbury, R. S. Nikhil, S. Peyton Jones, M. Reeve, and D. Wise. *Report on the Programming Language Haskell.* Technical Report, July 1988.