On the Automatic Parallelization of List-Based Functional Programs*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
email: schreine@risc.uni-linz.ac.at

July 1992

Abstract

We investigate the behavior of functional programs operating on list structures with respect to automatic parallelization. Many list-processing functions follow typical patterns of recursion that contain inherent *horizontal parallelism*, *data parallelism*, *vertical parallelism* and *stream parallelism*. Starting from a simple classification of list-processing functions, more and more general versions of functional patterns are presented and it is shown how they cover many concrete programs. The operational behavior of these functional patterns in a parallel environment is discussed and criteria are presented that allow the automatic classification of functional programs.

1 Introduction

A language is called *functional*, if a program in this language is a mathematical expression $f(a)$, i.e. the application of a function f to an argument a. The evaluation of $f(a)$ returns a result but has no effect on the execution of the rest of the program. During this process, in general several subexpressions are ready for evaluation. A main characteristics of functional languages is the fact that the total result is not influenced by the order in which these subexpressions are evaluated (provided that the evaluation terminates). In particular, all expressions may be evaluated *simultaneously* i.e. by parallel processes.

This property distincts functional languages (like ML [8] or HASKELL [6]) from imperative languages like FORTRAN or C where the execution order of statements is crucial. Basically, there are two main sources of parallelism in functional programs:

*Sponsored by the FWF-Project S5302-PHY "Parallel Symbolic Computation".

1. **Horizontal Parallelism:** On the computation of an expression $f(a,b)$, the subexpressions a and b may be simultaneously evaluated. The parallel computation of *independent argument expressions* is the main source of parallelism especially in all *divide and conquer* algorithms.

2. **Vertical Parallelism:** If an expression $f(g(a))$ is to be evaluated, the function invocations f and g may be active at the same time. This parallel execution of *nested function applications* makes sense especially if the result of f is a compound data structure (*producer/consumer* parallelism).

A functional program contains lots of function applications where literally *any* application (up to the level of basic operators) may be the source of horizontal or vertical parallelism. The main problem is that the sequential execution of a function is much faster than the creation of a separate process that also consumes a considerable amount of main memory. Hence, we cannot execute all function applications in parallel but have to select a small subset where parallelism really pays.

Currently, most approaches (e.g. [4]) base their decision whether to evaluate an expression by a new process or not on local criteria like the *grain size* of the expression (how long will its evaluation take?) or on *strictness properties* (how far must its result be known to continue execution?). Approximations for these quantities can be computed by program analysis techniques like *abstract interpretation* [3].

Automatically parallelizing compilers for imperative languages in general pursue a different strategy. Here the main approach is to recognize specific *patterns* that often arise in typical (scientific) programs and lend themselves for parallelization. Usually, these patterns are FORTRAN DO loops processing large arrays or matrices in some regular fashion for which parallelization strategies are known.

This paper (an extended abstract of [7]) investigates the possibilities of parallelization in functional programs operating over *linked lists*. Lists are the predominant kind of data structures in functional programming and many applications can be viewed as list processors. Starting from elementary list processing functions, we therefore develop more and more generalized versions and present examples that show how concrete programs fit into our framework. The style of this paper is rather informal i.e. no mathematical theory is developed for the automatic classification of programs according to the presented structures. However, we introduce our ideas with respect to parallelism and give criteria that may aid the compiler in order to recognize the structure and exploit this information in the compilation process.

2 Elementary List Processing

In this section, we will introduce several patterns that are a model for a wide class of list operations. All programs are presented in a notion that is common to most functional languages (for an introduction see [2]). Lists are objects of type

```
List(α) = nil | cons(α, List(α))
```

where α is a *type variable* i.e. a place holder for any concrete type.

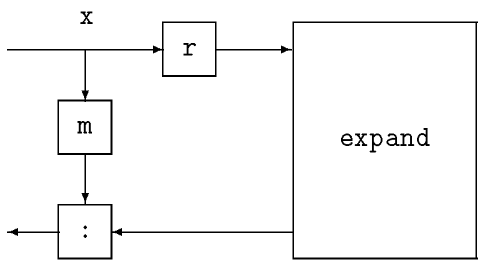
The execution model guiding our investigations is *dataflow* [1] which is an example of a *non-strict* evaluation model [9]: any function may immediately start execution in parallel; if for some operation the necessary data are not yet available, the process is temporarily blocked. Furthermore, the `cons` operator immediately returns as its result a list cell with empty head and tail slot (whose values will be computed by parallel processes). If some process tries to read such an empty slot, it is suspended.

2.1 Expansion

Lists must be some time generated from more elementary data. This task of *list expansion* is performed by functions of the pattern

```
expand :: α → List(β)
expand(x) =
  if t(x)
    then b(x)
    else cons(m(x), expand(r(x)))
```

that can be visualized by the following dataflow graph[1]:



where the dataflow node $\boxed{:}$ represents the `cons` operator, e.g.

```
squares :: Int → List(Int)
squares(n) =
  if n == 0
    then nil
    else cons(n*n, squares(n−1))
```

From the *parallelism* point of view, it is important to note that the operation `cons` is *not* strict in its arguments, i.e. the list cell can be constructed and returned as

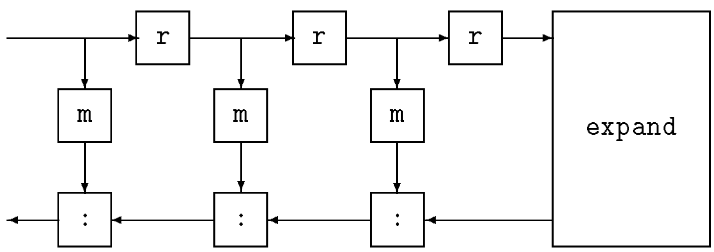
[1] All dataflow graphs in this paper show just the main (recursive) branch of the corresponding function.

the result before the the head or the tail of the list have been actually computed. The dataflow model therefore suggests the following parallel execution strategy:

1. First the termination condition `t(x)` is evaluated. If it becomes true, the result list `b(x)` can be generated immediately i.e. without recursion.

2. Otherwise, a new cell is allocated and returned. A new process is started that computes the head of the list and fills the head slot of this cell. In the meantime, the current process continues computation of the tail of the list.

Hence, the main process only constructs the "backbone" of the list, while the actual elements are computed by parallel processes (*data parallelism* which is a special case of *horizontal parallelism*). Moreover, since `expand` does not require the result of its recursive invocation to generate a new list cell, it actually delivers a "stream" of cells that other processes may simultaneously consume (*stream parallelism* which is a special form of *vertical parallelism*):



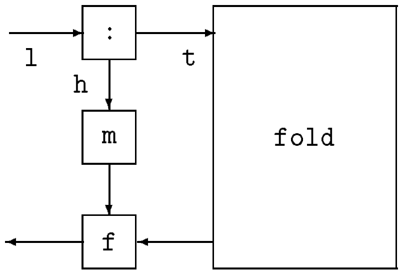
From this expansion of the dataflow graph, it becomes clear that each list element can be computed in parallel with the construction of the rest list and therefore with each other element of the list. The individual element processes are started in a rapid sequence that is "slowed down" by the evaluation of the various `b(x)`.

2.2 Folding

The inverse to list expansion is the *folding* operation that combines the various elements of a list into a single object (with a possibly different base type):

```
fold :: List(α) → β
fold(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒ f(m(h), fold(t))
```

The dataflow graph of this function is very similar to that of `expand`:



If the input list `l` is empty, some constant element `1` is returned. Otherwise, the actual fold function `f` combines the (modified) head of the list and the result of folding the rest list `t` into a single element, e.g.

```
sum :: List(Int) → Int
sum(l) =
  case l of
    nil ⇒ 0
    cons(h, t) ⇒ h + sum(t)
```

The essential difference between `expand` and `fold` is the fact that the folding operation `f` is usually *strict* in its arguments i.e. it can return a result only if both argument values are known. Consequently, the whole `fold` construct can deliver a result only if (at least a certain amount of information about) each element of its input list is available.

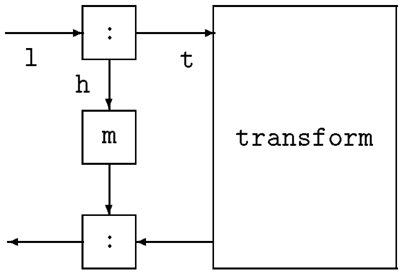
Analyzing the parallelism inherent in `fold`, it is obvious that, while the various element modifications `m(x)` (if there are any) can be computed in parallel, the actual folding performed by `f` are done in sequence "from right to left": first the tail list is folded and then the result is combined with the head of the list (this behavior is often called *reduce right*).

2.3 Transformation

The third basic operation is the *transformation* of one list into another:

```
transform(l) :: List(α) → List(β)
transform(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒ cons(m(h), transform(t))
```

The dataflow graph of `transform`



shows the similarities to `expand` and `fold`. However, in `transform` the result structure has the same complexity as the input structure, e.g.

```
mapf(l) =
  case l of
    nil ⇒ nil
    cons(h, t) ⇒ cons(f(h), mapf(t))
```

Again, `transform` itself just creates the structure of the result list, the lists elements are computed by parallel processes. However, in contrast to `expand` (and `fold`) the `transform` operator only requires the *structure* of its input to return (the structure of) its result list. If the actual input elements are not yet available, just the corresponding subprocesses are blocked. This is important since the execution of `transform` can *overlap* with the computation of its input, i.e. a list transformer is no barrier for vertical parallelism!

A special case of list transformation is the element-wise *merging* of two lists where both lists are processed element by element and "zipped" into one. Dynamically, this `merge` pattern behaves similar to `transform` except that synchronization is now performed on *two* input lists: the structure creation process can overlap (but must be synchronized) with the structure creation of both input lists and the element computing processes are blocked if the corresponding input elements are not available. For details, see [7].

2.4 Summary

In this section, we have presented several patterns of list processing functions for which the following features were common;

1. The list-generating functions return their results as *streams* of list cells, i.e. the consumption of these lists can overlap with their creation. Moreover, the computation of the individual list elements can be performed by parallel processes.

2. The list-consuming functions process the structure of their input lists in a regular way (independent of the actual element values). The execution of these functions can therefore overlap with the construction of their arguments.

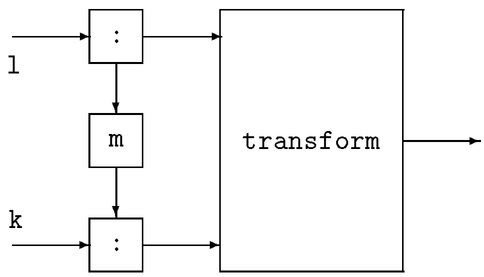
Based on these functions, *pipelines* of processes can be created that are connected by streams and work in parallel.

3 Tail-Recursion

List processing functions are often written in a tail-recursive style that is fundamentally different from that presented in the previous section. E.g. list transformation may be also formulated as follows

```
transform :: (List(α), List(β)) → List(γ)
transform(l, k) =
  case l of
   nil ⇒ b(k)
   cons(h, t) ⇒ transform(t, cons(m(h), k))
```

and the dataflow graph looks like



Tail-recursive list transformers are of interest because the generated list is essentially a modified version of the *reversed* input list i.e. the head of the result list stems from the last element of the input list. This function differs from the original solution in that an auxiliary parameter holds at any stage of the computation that part of the result list that has been constructed until now. Only when the base case of recursion is reached, the result is actually delivered to the caller.

Usually this extra parameter is hidden by a "shell" function that provides its initial value (`nil` for list-generators). However, we may also pass the result list of another function as the initial argument and hence perform a *constant time* concatenation of lists! This technique is the basis of several efficient divide and conquer algorithms with *quasi-horizontal parallelism* (see Section 7).

Operationally, in tail-recursive list generators the structure of the result list becomes "visible" to the caller only when the base case of recursion has been. Consequently, only finite list structures can be generated in a tail-recursive way (while in non-strict functional languages potentially indefinite lists are possible).

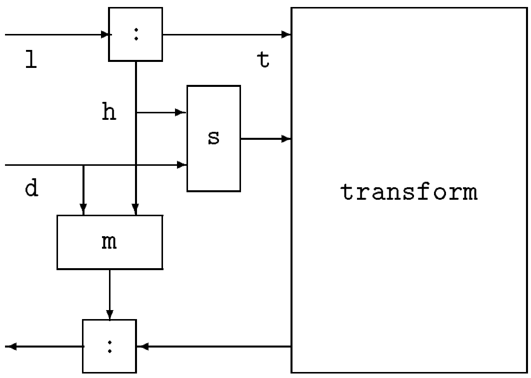
Moreover, the creation of the list structure can not overlap with its consumption, which makes the lists look like compact "arrays" (in contrast to "streams") and thus prevents stream parallelism. However, the *horizontal parallelism* in the computation of the individual list elements is not affected as well as *vertical parallelism* between processes that compute the individual list elements and processes that consume these elements.

4 Downward Communication

The list consuming functions of the previous sections were very restricted in the sense that, at any level of recursion, computation was only driven by one element (the current head) of the input list. In particular, no information about the preceding (and already traversed) part of the list could be transmitted between different recursive function activations. In this section, we loosen this restriction by introducing an additional argument that controls the computation. By this modification, e.g. the `transform` pattern changes to

```
transform :: (List(α), β) → List(γ)
transform(l, d) =
  case l of
    nil ⇒ b(d)
    cons(h, t) ⇒ cons(m(h, d), transform(t, s(h, d)))
```

with the following dataflow graph



The modifier `m` is now not only driven by the current list head `h` but also by the extra argument `d`. `s` computes from `h` and `d` the successor argument that is passed to the next recursive function invocation. Hence, information about the already processed part of the list may be propagated "downwards" the levels of recursion.

By unfolding the recursive structure of this graph, it becomes clear that each modification `m` now depends on the results of all previous `s` operations which may reduce parallelism between the different activations of `m`. In particular, if `s` and `m` are combined to one function, the whole `fold` construct becomes essentially serial. Only if `m` and `s` are independent and `m` takes significantly longer time than `s`, parallelism is preserved. An example for a downward communicating merger is

```
add2LD(l1, l2, d) =
  case (l1, l2) of
    (_, nil) ⇒ addLD(l1, c)
    (nil, _) ⇒ addLD(l2, c)
    (cons(h1, t1), cons(h2, t2)) ⇒
      let (s, c) = add3D(h1, h2, d) in
      cons(s, add2LD(t1, t2, c))
```

that adds two arbitrary precision integers represented by lists of fixed-size words. Execution is basically sequential since `add3D` computes `s` and `c` "atomically".

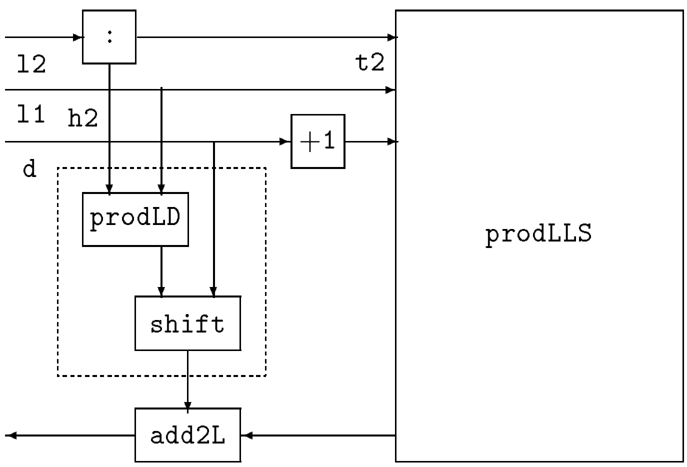
It is important to note that the flow of information controlling the computation in the current recursion level may also run from the recursive invocation to the caller. By this *upward communication*, a data dependency is created that may serializes the computation in the other direction. In an extreme case, the list elements may be processed sequentially from the tail of the list to its head! For details, see [7].

5 Generalized List Construction

Up to now, we only considered very special forms of list generating functions, namely those where the constructor operation `cons` directly appeared in the recursive branch. Now take a look at the following function:

```
prodLLS :: (List(Int), List(Int)) → List(Int)
prodLLS(l1, l2, d) =
  case l2 of
    nil ⇒ cons(0, nil)
    cons(h2, t2) ⇒
      add2L(shift(prodLD(l1, h2), d), prodLLS(l1, t2, d+1))
```

with dataflow graph



`prodLLS` multiplies two long (arbitrary precision) integers `l1` and `l2` and shifts the result by `d` positions. This function has a structure that is very similar to that of a `transform` operation: the input list `l2` is decomposed into its head `h2` and its tail `t2`. `t2` is fed into the recursive activation of `prodLLS` while `h2` is input to a compound expression with applications of the recursive functions `shift` and `prodLD`. The result of both subexpressions (i.e. integer lists) are merged by a function `add2L`. The function arguments `l1` and `d` are (almost unchanged) communicated in the downward direction.

The function therefore has a generalized `transform` structure (neglecting the downward communication)

```
transform(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒ C(h, transform(l))
```

where `C` is an expression computing from the head `h` and the result list of the recursive `transform` invocation a new result list. In the previous sections, we only considered cases where `C(h,t)` was defined as `cons(m(h), t)` but from the example above we see that it may also make sense to allow more general forms of construction.

The actual behavior of the function depends on the concrete version of `C`, but if we assume that `C` does not immediately require all of its list argument to continue execution, the dataflow behavior shows strong similarities: for all elements of the input `l`, `C` processes are started that can execute in parallel. In particular, if we can decompose into a "constructor" `c` and a "modifier" `m` that does *not* depend on `l`, the individual instances of `m` can execute in parallel Often, there is also vertical parallelism between the individual instances of `c`, the amount of this parallelism depends on the "strictness" of `c` in its second argument.

6 Conditional List Consumption

The list consuming functions presented up to now processed their input lists in a very regular fashion, i.e. each element of the input list was treated in the same way. In particular, the recursive activation of a function could be created as soon as the structure of the input list was available but *before* the actual element values were known. In this section, we will investigate the behavior of list consumers in the case that a condition involving the current list head has to be evaluated, before the recursive function activation can be created. E.g. take the pattern

```
transform(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒ if p(h) then t1(h, l, d) else t2(h, l, d)
```

The recursive branch of the `transform` body is now enclosed by a test predicate `p` that depends on the value of the list head `h` and selects one of two branches for further evaluation. Assuming that `p` is strict in `h`, execution is therefore blocked until the corresponding value has been computed by a parallel process. A condition in a list transformer is therefore a *barrier* for vertical parallelism since the result structure can be constructed only if the elements of the input list are available. A typical example for conditional list transformation is the higher order function

```
filter :: (List(α), α → Bool) → List(α)
filter(l, p) =
  case l of
    nil ⇒ nil
    cons(h, t) ⇒
        if p(h)
          then cons(h, filter(t, p))
          else filter(t, p)
```

that takes a list and a predicate and returns as its result a version of the input list where all elements have been removed for which the predicate does not hold. It should be noted that the barrier for vertical parallelism only occurs if the recursive function calls occurs *within* a branch of a conditional expression. Recursive function calls should therefore be drawn outside conditionals whenever possible. E.g. if we replace the recursive branch of the `filter` body by

```
let r = filter(t, p) in
if p(h)
  then cons(h, r)
  else r
```

all elements of the input list can be processed in parallel.

7 Divide and Conquer

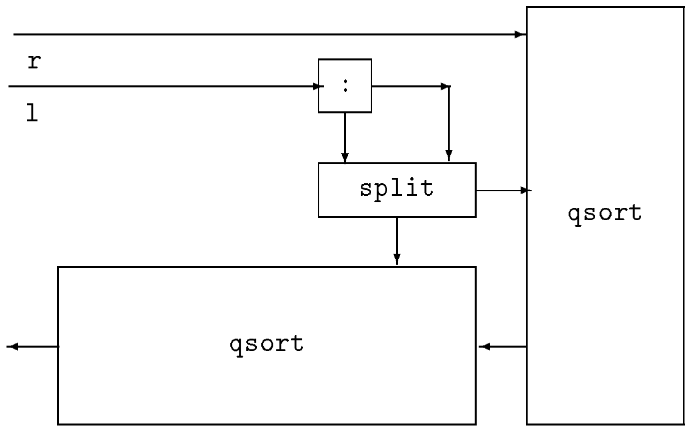
Since the type definition of lists contains only one recursive substructure (the list tail), list-processing functions do usually not lend themselves for parallelization based on the divide and conquer approach. In particular for list generators, the results of two recursive function applications must be in some way combined to form a longer list. Since this combination process is usually linear in time, one can in most cases find more efficient solutions than divide and conquer.

However, there is one method to combine lists in constant time: the implicit concatenation by an extra argument to the generator of the first sublist (see Section 3). This technique that is often applied in functional programs also is the origin of several efficient divide and conquer programs.

Let us consider the functional version of the Quicksort algorithm (with an extra argumnet for list concatenation):

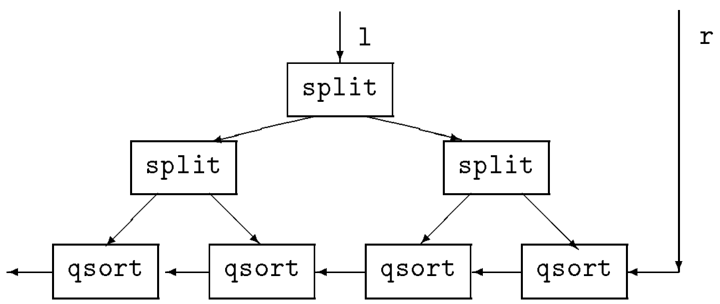
```
qsort :: (List(α), List(α)) → List(α)
qsort(l, r) =
  case l of
    nil ⇒ r
    cons(h, t) ⇒
      let (a, b) = split(h, t) in
      qsort(a, qsort(b, r))
```

The pattern of the program body of and the structure of its dataflow graph



shows that `qsort` is just a list transformer where the constructor function happens to be the function itself. Since the second argument of `qsort` is is just forwarded, both recursive invocations can execute independently. This is the main trick of many list-based functional programs based on the divide and conquer approach: the horizontal parallelism between independent recursive function invocations (followed by an explicit `append` operation) is replaced by vertical parallelism between nested applications (i.e. one branch delivers its result to the other one).

Actually, for `qsort` the situation is a bit more complicated, since the whole input list is filtered by a function `split` that generates two sublists `a` and `b` that are passed to the recursive function applications, respectively. By expanding the double-recursive structure of the dataflow graph, we get a tree of processes where the leaves receive and concatenate the trivial one-element lists (which are already sorted):



From the parallelism point of view, `qsort` is a therefore a perfect divide and conquer program (where the whole task is performed in the divide phase of computation while the combination process itself is trivial).

8 Automatic Classification and Parallelization

The patterns presented in this paper provided some abstraction that helped to clarify the parallel features of typical list-based programs. However, in concrete applications these patterns may appear in different forms and variations. Due to lack of space, we can only sketch the techniques that may be used to detect the typical feature of a say list-transformer in a concrete function (for more details, see [7]):

1. **Strictness Analysis** deals with the question "to which extent must x be known such that a certain amount of $f(x)$ can be computed?". By *abstract interpretation* [3], an answer to this problem can be given and the "strictness interface" of functions can be approximated. Some patterns presented in this paper have characteristic strictness interfaces such that one can estimate the pattern from this interface without "looking" inside the body. However, often this analysis is too weak to express the dynamical behavior of a function.

2. **Structure Analysis** looks inside the body to determine the "recursion structure" of the function. By focusing on the constructs that are important for parallelization, an abstracting view of the function is provided. These crucial constructs are case expressions, conditionals, `cons` operators and recursive function applications. The result of the analysis is a tree of evaluation pathes where the inner nodes correspond to case constructs or conditionals and the leaves are the possible result values.

3. **Dataflow Analysis** investigates the dependencies between the points determined by structure analysis. Each input list is analyzed with respect to its application (whether is it matched in a case statement, used in a condition or just forwarded to other functions); the same analysis is performed for the results of recursive function activations. We then can determine whether lists are constructed "stream-wise" or "array-wise", whether the individual list elements can be computed independently or not and in which direction data dependencies between a function and its recursive activation do exist.

In a top-down analysis phase, this information can be combined to detect the outermost list-processing function applications where internal parallelization (data parallelism and vertical parallelism) is possible and achieves sufficiently large granularity. Independent list-processing function applications may be executed in parallel (*horizontal parallelism* or *stream-parallelism*). This classification process may be also essentially simplified by providing *higher-order* functions that correspond to functional patterns and may be applied by the user to express parallelization points in a program (this leads to the idea of *para-functional programming* [5]).

9 Conclusions

The main idea expressed in this paper is that the automatic parallelization of functional languages should concentrate on typical patterns that often occur in list-processing functions (this idea can be generalized for other recursively defined data structures e.g. trees as well). By automatic analysis techniques, the typical behavior of a function can be classified according to various criteria that express the dynamic behavior of the program in a parallel environment. The author's ongoing Ph.D. thesis aims to formalize these ideas and make them suitable for the actual implementation in an automatically parallelizing compiler.

References

[1] Arvind and R. S. Nikhil. Executing a Program on the MIT Tagged-Token Dataflow Architecture. PARLE 87, Conference on Parallel Architectures and Languages Europe, Eindhoven, The Netherlands, June 15–19, 1987, Volume 259 of Lecture Notes in Computer Science, Springer, Berlin.

[2] Richard Bird and Philip Wadler. *Introduction to Functional Programming.* Prentice-Hall International Series in Computer Science, Englewood Cliffs, New Jersey, 1988.

[3] Geoffrey Burn. *Lazy Functional Languages: Abstract Interpretation and Compilation.* Research Monographs in Parallel and Distributed Computing. Pitman, London, 1991.

[4] Benjamin F. Goldberg. Multiprocessor Execution of Functional Programs. Research Report YALEU/DCS/RR-618, Department of Computer Science, Yale University, April 1988.

[5] Paul Hudak. Para-Functional Programming. *IEEE Computer*, 19(8):60–70, August 1986.

[6] Paul Hudak, Simon Peyton Jones, Philip Wadler et al. Report on the Functional Programming Language Haskell. Research Report 1.1, Department of Computing Science, University of Glasgow, Glasgow, UK, June 1991. Final Draft.

[7] Wolfgang Schreiner. On the Analysis of List-Based Functional Programs for Automatic Parallelization. Technical report, RISC-Linz, Johannes Kepler University, Linz, Austria, 1992. To appear.

[8] A. Wikström. *Functional Programming Using Standard ML.* International Series in Computer Science. Prentice-Hall, 1987.

[9] S. C. Wray and J. Fairbairn. Non-strict Languages — Programming and Implementation. *The Computer Journal*, 32(2):142–151, 1989.