# On the Analysis of List-Based Functional Programs for Automatic Parallelization*

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

May 4, 1993

**Abstract**

We investigate the analysis 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 into *expanders*, *folders*, *transformers* and *mergers*, 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. Finally, it is sketched how a compiler may exploit this information for automatic parallelization and code generation.

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

# 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. A functional program contains a set of user-defined functions that set up an environment in which the actual program expression can be evaluated. 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 [Wikström, 1987] or HASKELL [Hudak *et al.*, 1991]) from imperative languages like FORTRAN or C where the execution order of statements is crucial. It also is the reason why functional languages are often said to be better suitable for the programming of parallel computers. In the last decade, many research groups have started to work on the automatic parallelization of functional languages but the initial enthusiastic expectations could not yet be fulfilled. Today parallel computers are (with few exceptions) still programmed in the "old-fashioned" imperative style of programming.

Before we explain this gap between theory and praxis of parallel functional programming, let's take a look on the two main sources of parallelism in functional programs:

1. **Horizontal Parallelism:** If the result of an expression $f(a, b)$ is to be computed, the subexpressions $a$ and $b$ may be simultaneously evaluated. The parallel evaluation 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 if $f$ not immediately needs the result of $g$ to proceed execution or if $g$ generates 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.

The problem of *selection* also arises in the automatic parallelization of imperative languages but in some very different sense: here those parts of a program have to be identified that may be executed in parallel without changing the result. In functional languages, we already know that any part may be executed in parallel. However, from the large set of possibilities we have to identify those ones where parallelism really helps to decrease the runtime of the program!

Currently, most approaches (e.g. [Goldberg, 1988]) base their decision whether to evaluate an expression sequentially (by the current process) or in parallel (by a new process) 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*: the functional program is translated into an abstract domain that expresses the interesting runtime property of the program. The compiler then executes the abstracted program and applies the generated results to optimize and to parallelize the program code.

However, the information provided by abstract interpretation is quite vague and still leads to too fine-grained and therefore inefficient parallel programs. A main research direction in this area is therefore to improve the mathematical toolkit in order to receive more accurate information (e.g. by *projections* [Wadler and Hughes, 1987]).

Automatically parallelizing compilers for imperative languages in general pursue a different strategy. Of course, complex mathematical techniques are applied here, too. But 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.

On the contrary, in the area of functional languages programs are usually considered as amorphic heaps of function applications without recognizing the special structures that can be found in most of them. By concentrating on the efficient parallelization of these structures, probably better results could be produced for many typical applications.

In this paper, we investigate the parallelization of one particular kind of functional programs, namely programs operating over *linked lists*. Lists are the predominant form of data structures in functional programming and many applications can be viewed as list processors. Moreover, we believe that the only fruitful parallelism in functional programs actually stems from the parallel processing of data structures. 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 types of program structures. However, we introduce our ideas with respect to parallelism and also give some criteria that may aid the compiler in order to recognize the structure and exploit this information in the compilation process. The underlying program structures can be also found in Guarded Horn Clause Languages (e.g. PARLOG [Clark and Gregory, 1984] or STRAND [Foster and Taylor, 1989]) that are in many aspects very similar to functional languages.

# 2 Elementary List Processing

In this section, we will introduce four types of functions that all process lists in a fundamentally different way and 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 [Bird and Wadler, 1988]).

Lists in functional languages are objects of type

$$\texttt{List}(\alpha) = \texttt{nil} \mid \texttt{cons}(\alpha, \texttt{List}(\alpha))$$

where $\alpha$ is a *type variable* i.e. a place holder for any concrete type (e.g. integers, booleans, lists of integers, ...).

This definition can be read as follows: any object of type $\texttt{List}(\alpha)$ is either the constant `nil` (the empty list) or a composition of an object of type $\alpha$ (the *head* of the list) and an object of type $\texttt{List}(\alpha)$ (the *tail* of the list).

The execution model guiding our investigations is *dataflow* [Arvind and Nikhil, 1988], i.e. any function may immediately start execution in parallel. If for some operation the necessary data are not yet available, the corresponding 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. Hence, dataflow is an example of a *non-strict* model for the evaluation of functional programs [Wray and Fairbairn, 1989].

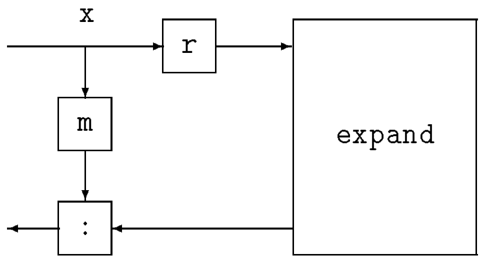
We denote as *list consumers* functions that have lists as input arguments and traverse these lists to compute their results. *List generators* are all functions that produce new output lists. *(Functional) patterns* are function definitions with placeholders for the application of concrete functions. We use the same syntax for patterns as for concrete functions, the distinction will be clear from the context.

## 2.1 Expansion

Lists must be somewhen generated from data that are more elementary than lists. This task of *expanding* elementary objects into lists is performed by functions of the pattern

```
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. The parameter `x` represents all input data that determine the new list. The construction process resembles the recursive nature of lists:

1. If the termination condition `t(x)` is true, a "base list" `b(x)` is created and returned as the result. `b(x)` is usually the constant `nil` but it may be also any more complicated expression that does not contain any recursive call of `expand`.

2. If `t(x)` is false, a new list cell is created whose head is computed by the modification function `m` and whose tail is the result of the recursive application of `expand` to the "rest" `r(x)`.

One example for a function belonging to the class `expand` is

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

returning the list of the first $n$ squares. Also the following function belongs to this type

```
sublists :: List(α) → List(List(α))
sublists(l) =
  case l of
    nil ⇒ cons(nil, nil)
    cons(h, t) ⇒ cons(l, sublists(t))
```

that creates the lists of all sublists of a given list `l`.

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

In general, `expand` is typed

$$\texttt{expand :: } \alpha \rightarrow \texttt{List}(\beta)$$

where $\alpha$ and $\beta$ are arbitrary types with the restriction that both have the same "nesting level" of `List` constructions: if $\alpha$ is some type $\texttt{List}^n(\gamma)$, then $\beta$ is a type $\texttt{List}^n(\delta)$ and vice versa (where $\gamma$ and $\delta$ do not contain any `List` constructors on the top level). In other words, `expand` adds one level of `List` nestings to its argument (with a possible transformation of the base type).

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 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:

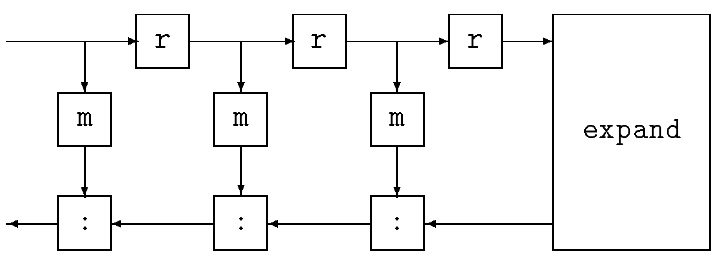
1. First the termination condition `t(x)` is evaluated. If it becomes true, the result list `b(x)` can be generated immediately i.e. without recursion.
2. Otherwise, a new list cell is allocated and returned as the result of the function. A new process is started that will compute the head of the list and somewhen fill the empty head slot of the cell. In the meantime, the current process iteratively continues execution computing 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*). 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*).

The analysis of data parallelism (which is a special case of *horizontal parallelism*) and of stream parallelism (which is a special case of *vertical parallelism*) is the core of our investigations, i.e. we aim to answer the questions:

1. How can the list elements be computed in parallel?
2. How can the list construction overlap with its consumption?

The behavior of `expand` is more intuitvely described by expanding the recursive structure of the dataflow graph presented above:



From this picture, 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)` and `r(x)`.

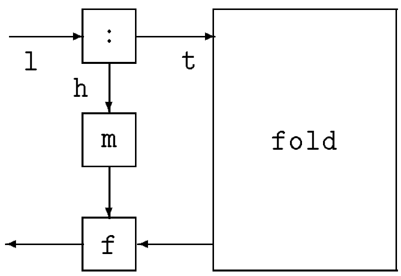
## 2.2 Folding

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 type $\beta$ has the same nesting level of `List` constructions as $\alpha$, i.e. `fold` removes one `List` level from the type of its input argument.

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




If the input list `l` is empty, some constant element `l` 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.

Examples for folding operations are the computation of the sum of a list of integers

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

or the "flattening" of a list of list by appending the individual element lists:

```
flatten :: List(List(α)) → List(α)
flatten(l) =
  case l of
    nil ⇒ nil
    cons(h, t) ⇒ append(h, flatten(t))
```

The essential difference between `expand` and `fold` is the fact that in contrast to the constructor operation `cons` the folding operation `f` is *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*).

Folding operations in functional languages are therefore bottlenecks for parallelization: while the folding of a list can usually overlap with its construction (*vertical parallelism*), the folding process itself is mainly sequential. This is even true for commutative and associative operations like integer addition where the order of operations would have no influence on the result.

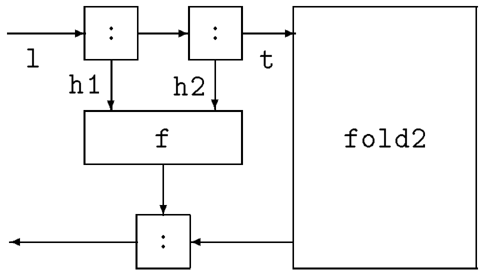
However, since functional languages are *deterministic* (i.e. always the same set of operations is performed), this additional knowledge can be usually not exploited. One workaround to this problem would be to restructure the program in the following way

```
fold :: List(α) → α
fold(l) =
  case l of
    nil ⇒ b1
    cons(h, nil) ⇒ h
    cons(_, cons(_, _)) ⇒ fold(fold2(l))
```

whose expanded dataflow graph is just a pipeline of applications of the auxiliary function

```
fold2 :: List(α) → List(α)
fold2(l) =
  case l of
    nil ⇒ l
    cons(_, nil) ⇒ l
    cons(h1, cons(h2, t)) ⇒ cons(f(h1, h2), fold2(t))
```

with the following dataflow graph:




`fold2` is just a special list transformer (see the next subsection) that returns a result list whose length is half of the length of the input list. The elements of the result list are computed by parallel `f` processes that combine two successive elements of the input list, respectively. The result of `fold2` is recursively fed into `fold` until a list with a single elment is derived that is returned as the result of the `fold` operation.

Thus the elements are folded in a tree-like fashion where each level of the tree is represented by the input list of one `fold` process. This solution applies as well *horizontal parallelism* within one level of the tree as *vertical parallelism* between different levels (if some folding operations take longer than others).

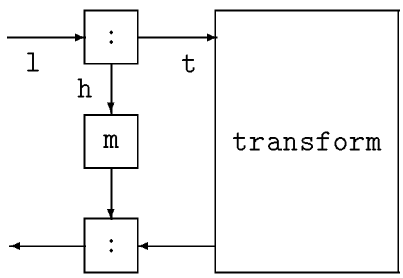
This way of parallelizing the `fold` operation is possible in all cases where `f` is an associative operator on a monoid structure $\alpha$. However, it only pays if `f` is complex enough to justifice the parallelization overhead.

## 2.3 Transformation

The third basic operation is the *transformation* of one list into another in an element-wise fashion. List transformers are of the form

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

where the base types $\alpha$ and $\beta$ are on the same level with respect to `List` construction. 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. Examples for list transformations are all kinds of "map" functions like

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

that applies a function `f` to each element of the input list and returns the list of results.

Dynamically, `transform` behaves similar to `expand`: it iteratively creates the structure of the result list, the lists elements themselves 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 processes computing the actual element values `m(x)` are temporarily blocked.

This difference is especially important with respect to *vertical parallelism*: while `expand` is usually blocked if its input is not available, the execution of `transform` can *overlap* with the computation of its input, i.e. a list transformer is no barrier for vertical parallelism!

We will present more general (and more interesting) examples of list transformations in the following sections.

## 2.4 Merging

The fourth elementary operation on lists is the element-wise *merging* of two lists into one

```
merge :: (List(α), List(β)) → List(γ)
merge(l1, l2) =
  case (l1, l2) of
    (_, nil) ⇒ m1(l1)
    (nil, _) ⇒ m2(l2)
    (cons(h1, t1), cons(h2, t2)) ⇒ cons(m(h1, h2), merge(t1, t2))
```

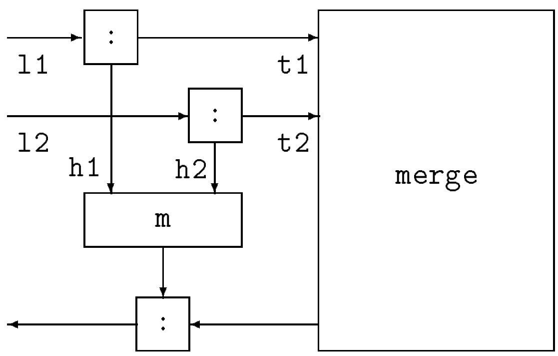
where $\alpha$, $\beta$ and $\gamma$ have the same level of `List` constructors. In `merge`, both lists are processed element by element and "zipped" into one. If one list is shorter than the other, a list transformer is applied to the not yet processed rest of the longer list.

One example for a merging functions is

```
add :: (List(Int), List(Int)) → List(Int)
add(l1, l2) =
    (_, nil) ⇒ l1
    (nil, _) ⇒ l2
    (cons(h1, t1), cons(h2, t2)) ⇒ cons(h1+h2, add(t1, t2))
```

that adds up two integer lists in an element-wise fashion.

Dynamically, the `merge` function 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 `m(h1, h2)` are blocked if the corresponding input elements are not available. This behavior is also illustrated by the following dataflow graph:




A `merge` function is therefore principially no barrier for vertical parallelism but the probability for blocking is much higher. More interesting examples for this kind of functions will be presented in the next sections.

## 2.5 Summary

In this section, we have presented four elementary types of list processing functions for the creation of lists, their folding, transformation and combination. Two features are common for these functions:

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.

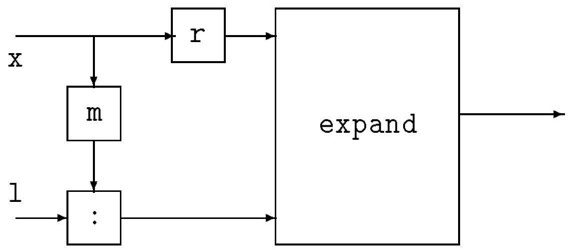
Based on these functions, *pipelines* of processes can be created that are connected by streams and work in parallel. The source of such a pipeline is always an `expand` operation, its sink is often a `fold` process.

# 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 expansion may be also formulated as follows

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

and the dataflow graph looks like




This function differs from the original solution in that an auxiliary parameter `l` has been introduced that 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. E.g. take the function

```
ints :: (Int, List(Int)) → List(Int)
ints(n, r) =
  if n == 0
    then nil
    else ints(n-1, cons(n, r))
```

that like the `ints` program of the previous section constructs a lists of integers but the elements are constructed in the reverse order! Often, an additional shell hiding this extra argument is wrapped around the function like in

```
intlist :: Int → List(Int)
intlist(n) = ints(n, nil)
```

For most applications, tail-recursive formulations are not very natural. However, there are three major motivations to write functions in this style:

1. **Efficiency:** Tail-recursive functions are considered to be more memory-efficient than their general counterparts, since a compiler may easily transform tail-recursion into iteration (where only one function activation is alive at a time). However, a clever compiler may also transform general list generating functions into a tail-recursive form (on the level of the imperative target code):

   The caller provides as an additional argument a pointer to a list cell with empty tail field. The callee allocates a new cell, writes its address into this empty slot, computes the head and calls itself tail-recursively with a pointer to the new cell. This transformation is always possible, if the tail of the list is not required for computing the head element of the list (i.e. no "upward communication" takes place, see Section 5).

2. **Element Order** The formulation of list generating functions sometimes becomes simpler if the list can be created in the reverse order i.e. starting with the last element. Also if lists have to be reverted for further processing, tail-recursive formulations are usually introduced (see the program `reverse` below).

3. **Concatenation** Usually the additional parameter for the accumulation of the result is hidden by a "shell" function. However, we can apply this extra parameter in order to concatenate the result list of another function with the actual result. Instead of introducing a linear time concatenation operation `append` as in

   ```
   f(n) = append(intlist(n), g(n))
   ```

   (where `intlist` is defined as above) we can simply write

   ```
   f(n) = ints(n, g(n))
   ```

   and so "implicitly" concatenate the results in constant time. This trick is well-known in functional programs and can be applied in most cases to avoid explicit `append` operations (also for not tail-recursive functions).

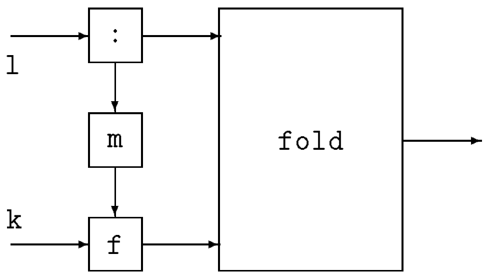
Operationally, the tail-recursive version of `expand` differs from the original solution in that the structure of the result list becomes "visible" to the caller only when `expand` has reached the base case of recursion. Consequently, only finite list structures can be generated in a tail-recursive way (while in non-strict functional languages potentially indefinite lists are possible). The creation of such a list structure can also not overlap with its consumption, which makes the lists more look like compact "arrays" (in contrast to the "streams" of the previous section).

However, this difference only influences *stream parallelism* with the caller of the function but not the horizontal parallelism in the computation of the individual list elements. Moreover, tail-recursive functions involved in producer-consumer parallelism may be more efficient since the consumer process cannot be blocked on the tail field of a list cell that the producer has not yet filled.

A tail recursive solution of `fold` has the following structure

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

visualized by the dataflow graph:




In contrast to the version in the previous section, this solution performs the folding operation from left to right, i.e. each element is directly combined with the next one. This behavior is also called *reduce left*.

One example for a tail-recursive `fold` operation is the function

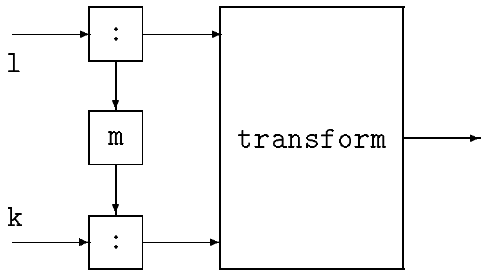
```
histogram :: (List(Int), Histogram) → Histogram
histogram(l, k) =
  case l of
    nil ⇒ k
    cons(h, t) ⇒ histogram(t, update(h, k))
```

that computes a histogram of the list `l`, i.e. it counts the number of occurrences of each integer in `l`. The auxiliary parameter `k` denotes the histogram (in some arbitrary representation) of that part of the list that preceedes the current head element `h`. `k` is updated wit respect to `h` and forwarded to the tail-recursive invocation that processes the rest of the list.

The tail-recursive solution for the `transform` function has the structure

```
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 its dataflow graph is depicted by



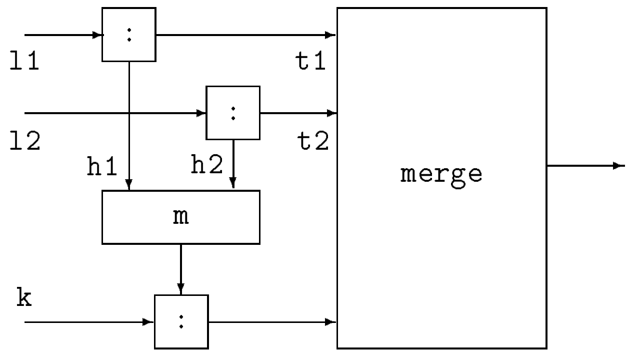

Tail-recursive list transformers are interesting because the generated list is essentially a modified version of the *reversed* input list i.e. the head of the result list stems from the last element of the input list. One example for a tail-recursive list transformer is the `reverse` function itself that is usually formulated as follows

```
reverse :: (List(α), List(α)) → List(α)
reverse(l, r) =
  case l of
    nil ⇒ r
    cons(h, t) ⇒ reverse(t, cons(h, r))
```

Finally, the tail-recursive version of `merge` has the following pattern

```
merge :: (List(α), List(β), List(γ)) → List(δ)
merge(l1, l2, k) =
  case (l1, l2) of
    (_, nil) ⇒ b1(l1, k)
    (nil, _) ⇒ b2(l2, k)
    (cons(h1, t1), cons(h2, t2)) ⇒
      merge(t1, t2, cons(m(h1, h2), k))
```

and the dataflow graph looks like




Summarizing, tail-recursive list generators (start processes that) produce the list elements in the reverse order and deliver the result structure in an "array-like" fashion (i.e. it becomes visible to the consumer only after the whole structure has been generated). Therefore, no stream parallelism in the structure generation is possible but still horizontal parallelism for the computation of the individual elements and vertical parallelism for the consumption of these elements. The tail-recursive `fold` reduces the input list from left to right and is due to tail-recursion more efficient than its "reduce right to left" counterpart.

# 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 of the input list. In particular, there existed no information about the preceding part of the list that was already traversed by the caller activations. In other words, no level of recursion was able to influence the next one, since the only information passed was the tail of the input list.

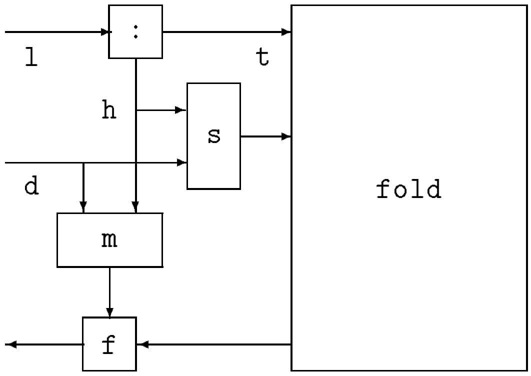
In this section, we loosen this restriction by introducing an additional argument that stands for all arguments that control the computation (besides the traversed list). We present this form just for the classical version of list processors but it is obvious how to extend it to the tail-recursive case.

By this modification, the `fold` pattern changes to

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

The modification function `m` is now not only driven by the current list head `h` but also by the extra argument `d`. The function `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 propagaged "downwards" the levels of recursion.

The dataflow graph of `fold` with downward communication looks as follows

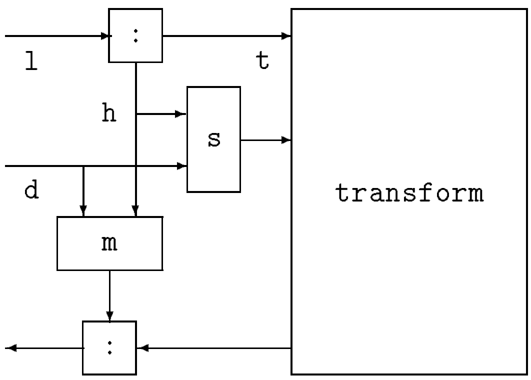



By unfolding the recursive graph structure, it becomes clear that each modification `m` now depends on the results of all previous `s` operations (assuming that `m` is strict in `d`). This new data dependency may reduce the parallelism among the activations of `m` and in extreme cases even serialize this process. 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.

The `transform` pattern changes by downward communication 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)))
```

The dataflow graph of `transform` resembles that of `fold`



except that the strict `f` is replaced by the non-strict `cons`. Dynamically, the same argument holds as for `fold`: if `m` and `s` are independent and `m` is significantly more costly than `m`, parallelism is preserved. In any other case, a certain amount of sequentialization takes place for the computation of the list elements (the construction of the list structure itself is not affected by this).

One example for a transformer with downward communication is the function `addLD` that adds an arbitrary precision integer (represented by a list of fixed-size integer digits) and a digit:

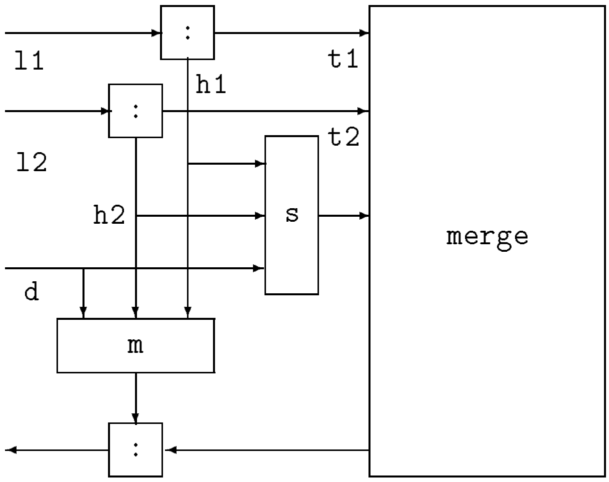
```
addLD :: (List(Int), Int) → Int
addLD(l, d) =
  case l of
    nil ⇒ cons(d, nil)
    cons(h, t) ⇒
      let (s, c) = add2(h, d) in
      cons(s, addLD(t, c))
```

`add2` adds two integer digits returning the sum `s` and the carry `c`. `s` represents the head of the result list while the carry is propagated to the recursive invocation of `addLD`. This function is sequentially executed since `s` and `c` are both computed by `add2` in an "atomic" fashion.

The pattern of `merge` with downward communication looks as follows

```
merge :: (List(α), List(β), γ) → List(δ)
merge(l1, l2, d) =
  case (l1, l2) of
    (_, nil) ⇒ b1(l1, d)
    (nil, _) ⇒ b2(l2, d)
    (cons(h1, t1), cons(h2, t2)) ⇒
      cons(m(h1, h2, d), merge(t1, t2, s(h1, h2, d)))
```

The dataflow graph of `merge`




shows that `merge` behaves similar to `transform`: the computation of the elements of the result list is seriously sequentialized unless `s` is a comparatively simple function independent of `m`.

A well-known example for a merging operation with downward communication, is the addition of two arbitrary precision integers represented by lists of fixed-size integer words:

```
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))
```

In `add2LD` (add two long integers and a digit) the extra argument `d` represents the carry of the previous additions. The head elements `h1` and `h2` are added to `d`, the sum `s` represents the head of the result list and the new carry `c` is propagated down the recursion. Again, this function is basically sequential since `s` and `c` are computed "atomically" (however, the creation of the result may vertically overlap with its consumption).

Summarizing, the main difference between a list consuming function with downward communication and a function where this communication does not take place is the fact that the former may propagate information about the already processed part of the list while the later may not. This additional data dependency may decrease the amount of parallelism between the different recursion levels up to a total serialization. However, in the next sections we will present several examples of functions with downward communication where much inherent parallelism is available.

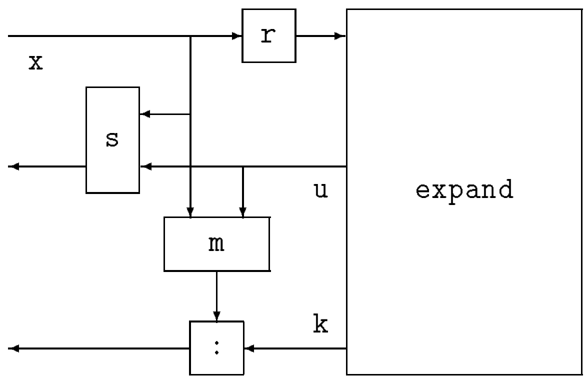
# 5 Upward Communication

Up to now, the generation of lists proceeded in such a way that the construction of the list tail had no influence on the computation of the list head. No information could be propagated from the recursive function activation to the caller (besides the result list written into the tail slot of the new list cell). By an additional generalization, this "upward" directed flow of information becomes possible. Each function can now deliver some control information that is used for processing the current head. This information is represented by an additional result value but in general it may also coincide with the result list itself.

The `expand` function in general looks then as follows

```
expand :: α → (β, List(γ))
expand(x) =
  if t(x)
    then (b1(l), b2(l))
    else
      let (u, k) = expand(r(x)) in
      (cons(m(x, u), k), s(x, u))
```

From the dataflow graph of `expand`




it is easy to see that the operational behavior of the function has essentially changed: while in the former version the current function activation only required information from the caller to compute a new list element, it now also requires information from the recursive activation, i.e. serialization takes place in the opposite direction.

E.g. in the following function

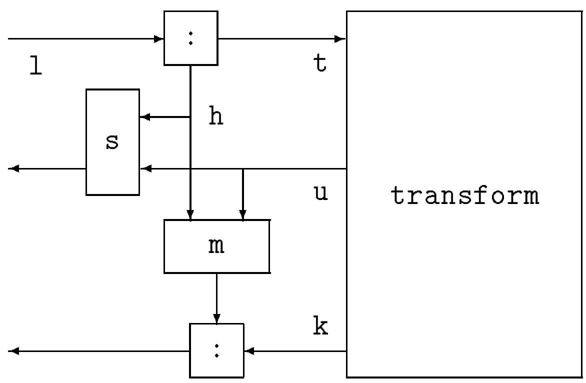
```
flist :: α → List(β)
flist(x) =
  if (term(x)) then
    cons(b(x), nil)
  else
    let r = fgen(s(x)) in
    cons(f(x, head(r)), r)
```

each element of the result list depends on the argument `x` and on the value of its successor element! In this extreme case, list elements can be only produced in sequence starting from the *end* of the list!

This change in program dynamics also affects the transformer function

```
transform :: List(α) → (β, List(γ))
transform(l) =
  case l of
    nil ⇒ (b1(l), b2(l))
    cons(h, t) ⇒
      let (u, k) = transform(t) in
      (cons(m(h, u), k), s(h, u))
```

with the dataflow graph




For example, take the function

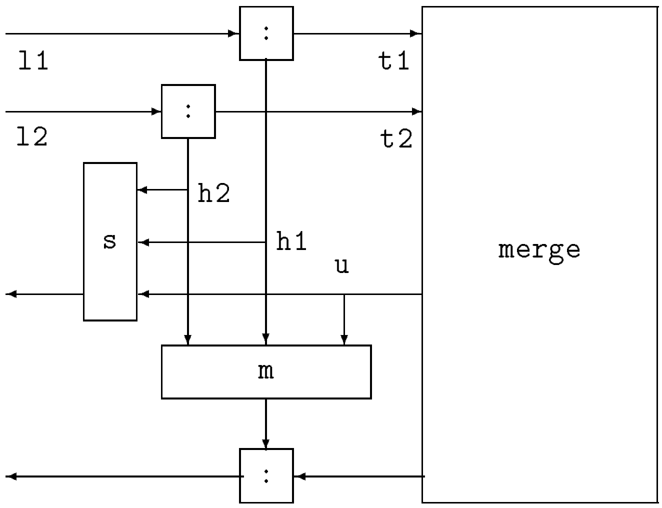
```
listsum(l) =
  case l of
    nil ⇒ (0, nil)
    cons(h, t) ⇒
      let (s, k) = listsum(t) in
      (cons(h+s, t), h+s)
```

that takes a integer list `l` and returns a list of integers whose $i$-th element is the element sum of the $i$-th tail list of `l`. This value is also returned as the first element of the result pair to the caller of the function. All elements are produced in sequence starting from the tail of the result list.

The same behavior takes place for the function `merge`

```
merge :: (List(α), List(β)) → (γ, List(δ))
merge(l1, l2) =
  case (l1, l2) of
    (_, nil) ⇒ (b11(l), b12(l))
    (nil, _) ⇒ (b21(l), b22(l))
    (cons(h1, t1), cons(h2, t2)) ⇒
      let (u, k) = merge(t1, t2) in
      (cons(m(h1, h2, u), k), s(h1, h2, u))
```

with dataflow graph



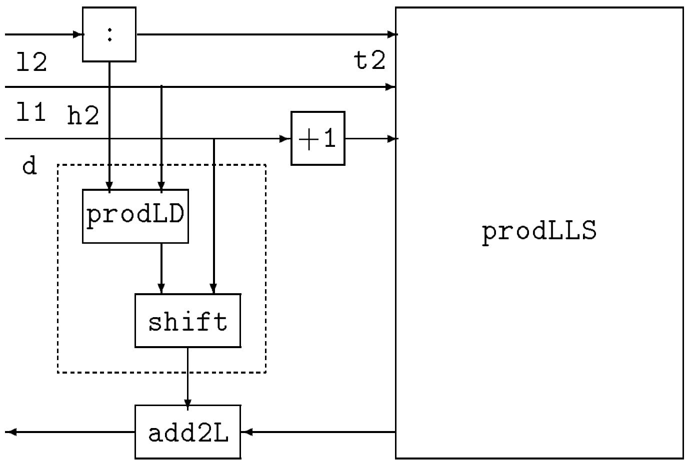

Summarizing, in list generators with upward communication the computation of each list element depends on information provided by the recursive invocation of the function. Depending on the (strictness) degree of this dependency, the various element-computing processes may be serialized from right to left. Anyway, list-generators with upward communication should be avoided whenever possible, since they cannot be automatically transformed into tail-recursive forms (as versions with only downward-directed communication).

# 6 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))
```

`prodLLS` multiplies two long (arbitrary precision) integers `l1` and `l2` and shifts the result by `d` positions. Long integers are represented by lists of fixed-size integer words where the head of the list denotes the least significant "digit" of the integer. This function has a structure that is very similar to that of a `transform` operation traversing the list `l2`:

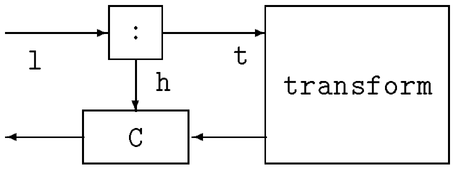



In case `l2` is empty, a constant list is returned. Otherwise, `l2` is decomposed into its head `h2` and its tail `t2`. `t2` is fed into the recursive activation of `prodLLS` while `h2` is input to a compound expression with applications of the recursive functions `shift` and `prodLD` (whose structure is analogous to that of the previously presented `addLD`). 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. The dataflow graph of this function pattern looks as follows:




In the previous sections, we only considered cases where `C` was defined as

```
C(h, t) = cons(m(h), t)
```

but from the example above we see that it may also make sense to allow more general forms of construction. In a similar way, the tail-recursive version of `transform` can be generalized

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

downward communication can be included

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

and also upward communication can be modelled

```
transform(l) =
  case l of
    nil ⇒(b1, b2)
    cons(h, t) ⇒
      let (u, k) = transform(t) in
      (s(h, u), C(h, u, k))
```

How far does the analogy between the original `transform` and its generalized version go with respect to parallelism? 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 `C` has the form

```
C(h, l) = c(m(h), l)
```

i.e. we can decompose `C` into a "constructor" `c` and a "modifier" `m` that does *not* dependend on `l`, the individual instances of `m` can execute in parallel (in `prodLLS`, `m` represented the composition of `shift` and `prodLD`). 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. In the above example, `c` was the list transformer `add2L` whose execution can overlap with the execution of all previous instances of `add2L` (stream parallelism).

This generalization of the constructor operation can be carried through for all other list generating function patterns in a similar way; we omit the details.

# 7 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 following variation of the `transform` pattern

```
transform(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒
      if p(h)
        then transform1(h, l, d)
        else transform2(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. A particular instance of `filter` is the function

```
split :: (α, List(α)) → (List(α), List(α))
split(e, l) =
  case l of
    nil ⇒ (nil, cons(e, nil))
    cons(h, t) ⇒
      let (a, b) = split(e, l)
      if h < e
        then (cons(h, a), b)
        else (a, cons(h, b))
```

that splits the input list `l` into two sublists by the pivot element `e`. It is easier to recognize the `filter` pattern, if one thinks of `split` as being composed of two separate `filter` functions where the first one filters all elements less than `e` and the second one all elements greater than `e` from the given list `l`.

In an analogous way, conditional versions of the functional pattern `merge`

```
merge(l1, l2) =
  case (l1, l2) of
    (_, nil) ⇒ b1(l1)
    (nil, _) ⇒ b2(l2)
    (cons(h1, t1), cons(h2, t2)) ⇒
      if p(h1, h2)
        then merge1(h1, h2, l1, l2)
        else merge2(h1, h2, l1, l2)
```

and of `fold` can be modelled

```
fold(l) =
  case l of
    nil ⇒ b
    cons(h, t) ⇒
      if t(h)
        then fold1(h, l)
        else fold2(h, l)
```

Please notice 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. take the following version of the `filter` function presented above:

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

While in the former solution each recursion level had to wait for the result of the test operation to proceed execution, this version can test all elements of the input list and construct the result list in parallel.

## Prime Number Generation

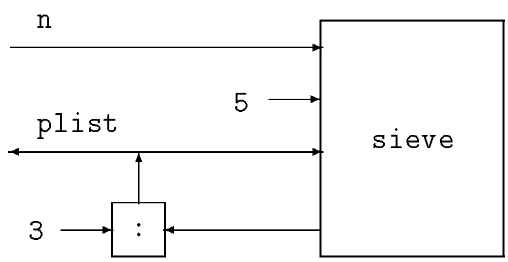
In the following, we present an interesting functional program that is essentially based on conditional list transformation (but in a very unusual way). The program `primes` solves the problem of computing all primes up to some bound $n$. The program is essentially based on a function `sieve` that receives the list of all primes less than some $t$ and returns a list of all primes from $t$ to $n$. In a naive approach, `sieve` would apply its input list to test $t$ and, if $t$ is actually prime, call itself tail-recursively with a copy of the input where $t$ has been appended. However, this copying process at any stage of the recursion would make the solution very inefficient.

Our version of `sieve` forwards to its recursive invocation the original list argument and returns as its result the list delivered by the recursive call with $t$ inserted at the *front*. How do then the recursive `sieve` applications ever get access to the new prime $t$? The crucial point of the program is in the function `primes` that calls `sieve` with its own result as its argument! Hence, each prime that is returned as the result of `sieve` is also available to all of the following invocations. The application of a *circular data-structure* (i.e. a structure that is defined in terms of itself) is only possible in a non-strict model of execution, where the arguments must not yet be fully evaluated when a function is applied.

The main function `primes` is just a shell around the circular definition of the data structure `plist`

```
primes :: Int → List(Int)
primes(n) =
  let plist = cons(3, sieve(n, 5, plist)) in
  plist
```

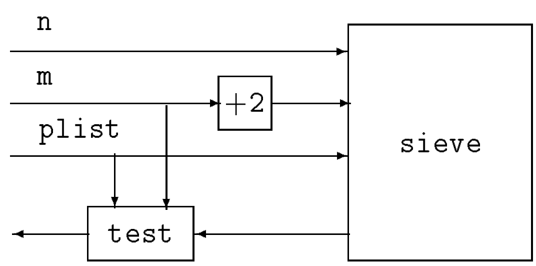
as illustrated by the following dataflow graph:




`sieve` is essentially a generalized `expand` function

```
sieve :: (Int, Int, List(Int)) → List(Int)
sieve(n, m, plist) =
  if n < m
    then nil
    else test(m, plist, sieve(n, m+2, plist))
```

where the result of the recursive application is fed into the constructor operation `test`:

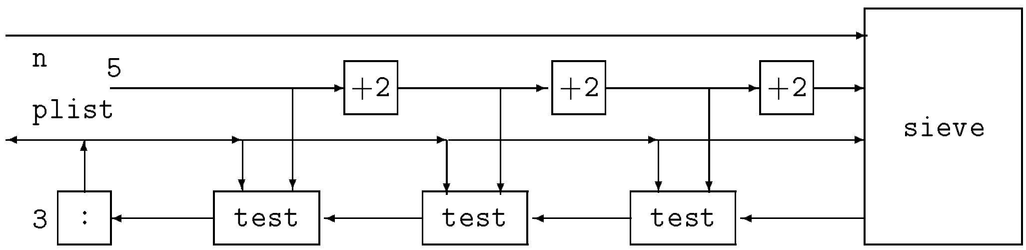



`test` itself is conditional transformer on the input `plist` with downward communication:

```
test :: (Int, List(Int), List(Int)) → List(Int)
test(m, plist, rest) =
  case plist of
    nil ⇒ cons(m, rest)
    cons(h, t) ⇒
      if m < h*h then
        cons(m, rest)
      else if m mod h == 0 then
        rest
      else
        test(t, plist, rest)
```

The test number `m` controls together with the head element `h` the construction process. `rest` is passed unchanged (and unused) through all recursion levels, the `test` function is therefore not strict in this argument.

The dynamic behavior of the program is best illustrated by the following expanded dataflow graph:




that shows the well-known pattern of list expansion with `test` as the generalized construction operation (remember that it is not strict in its parameter `rest`). This solution is totally based on vertical parallelism, since the individual `test` processes may execute independently and are only synchronized via the `plist` argument.

In general, many functional programs based on *sieve* algorithms have exactly the same structure as `primes`, e.g. the declarative version of Buchberger's *Gröbner Bases* algorithm in computer algebra [Siegl, 1990].

# 8 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.

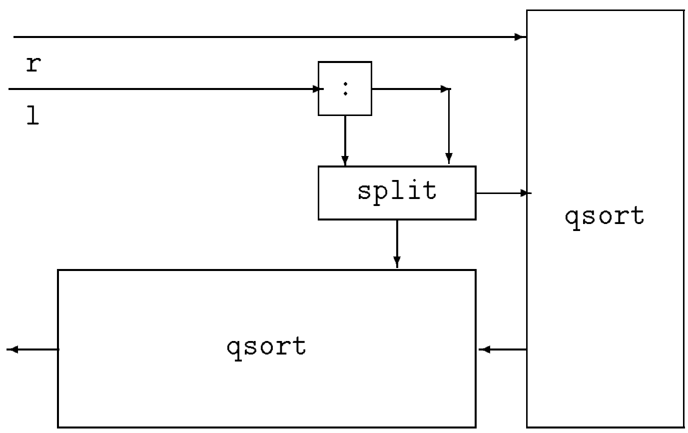
However, there is one method to combine lists in constant time: the implicit concatenation by an extra argument to the generator of the first sublist (see Section 3). This technique that is often applied in functional programs also is the origin of several efficient divide and conquer programs.

## 8.1 Quicksort

First, let us consider the functional version of the Quicksort algorithm operation on lists. By the introduction of an extra argument with which the result list is to be concatenated, we get the following program:

```
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))
```

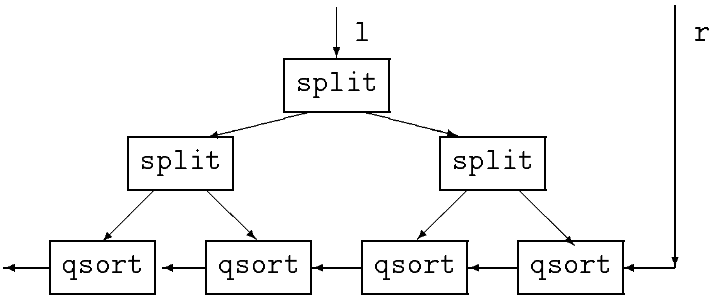
The pattern of the program body of and the structure of its dataflow graph




make clear that `qsort` is just a general-version of a tail-recursive `transform` where the constructor function happens to be a recursive call of the function itself. Since `qsort` is non-strict in its second argument (it is passed through all recursive invocations), both recursive invocations can execute independently. This is the main trick of all efficient 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. `split` is just the conditional list transformer presented in Section 7.

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).

## 8.2 Queens Problem

Finally, we present some `expand` like program based on the divide and conquer approach: the task is to write a function `queens` that takes a number $n$ and returns the list of all solutions of the $n$-queens problem, i.e. all positions of $n$ queens on a chess board of size $n * n$ such that no queen can beat another one. This example is one out of a class of search space problems that is in sequential languages usually solved by the backtracking technique.

Our solution applies a function `solve` that takes the actual board position *col* and *row* and a partial solution *part* consisting of *row−1* queens. By two recursive calls, `solve` computes all valid solutions starting with *part* and extended by queens at positons *col* to *n*: The first call computes all solutions starting with *part* and extended by queens at positions *col+1* to $n$ (i.e. the search tree is traversed horizontally). The second recursive call receives the partial solution *part* extended by the actual queen position *col* and computes all solutions starting with this longer partial solution of *row* queens (i.e. it steps down one level in the search tree). The result of one recursive call is fed to the other call in order to avoid the explicit concatenation of lists.

The top-level function

```
Board = List(Int)
queens :: Int → List(Board)
queens(n) = solve n 1 1 nil nil
```

is just a shell around the actual program

```
solve :: (Int, Int, Int, Board, List(Board)) → List(Board)
solve(n, col, row, part, rest) =
  if col > n then
    rest
  else
    let sols = solve(n, col+1, row, part, rest) in
    if conflict(col, 1, part) then
      sols
    else if row == n then
      cons(cons(col, part), sols)
    else
      solve(n, 1, row+1, cons(col, part), sols)
```

that is a tail-recursive function of type `expand` with an accumulator argument `part` and an auxiliary argument `rest` that is only used for concatenation. By unfolding the recursive structure of the function we receive an $n$-ary tree of processes where the leaves concatenate the received results (similar to the quicksort example in the previous subsection).

`conflict` is a tail-recursive version of type fold `fold` that traverses a partial solution `sol` and checks whether each queen position is compatible with the new position:

```
conflict :: (Int, Int, List(Int)) → Bool
conflict(col, dist, sol) =
  case sol of
    nil ⇒ False
    cons(h, t) ⇒
      beats(col, dist, h) or conflict(col, dist+1, t)
```

The `or` function performs the folding operation while the actual test operation `beats` is the element modifier.

# 9 Automatic Classification and Parallelization

## 9.1 Classification Techniques

The functional patterns `expand`, `fold`, `transform` and `merge` provided some abstraction that was necessary to clarify the parallel features of functional programs by neglecting all less important details. However, in concrete applications these patterns will not appear in the pure form presented in the previous sections but a function will often combine the features of different patterns (on different arguments). The question is now which tools can be applied to find the abstract behavior of a say "list transformer" in a concrete function.

### 9.1.1 Pattern Matching

A straight-forward implementation of the idea of list-processing patterns would be to *match* the actual syntactic structure of a program against these patterns in order to detect the *intension* of the program and thus draw conclusions about the dynamical behavior of the program. The drawback of this approach is that the concrete program syntax may differ considerably from the idealized pattern which makes the recognition process very difficult. However, if we apply pattern matching to characteristic substructures of a function body (e.g. `case` constructs or `cons` expressions), the approach may become feasible.

### 9.1.2 Higher Order Functions

The only difference between functional patterns and actual function definitions is the instantiation of the placeholder functions in the pattern by concrete function applications. Hence, it is possible to abstract the patterns into *higher-order functions* by additional arguments that represent the basic functions to be applied. The human programmer applys these higher-order functions to concrete functional arguments (the classification process becomes trivial then). Moreover, we can provide sequential and parallel versions of these higher order functions and let the user explicitly select those cases where he wants to utilize parallelism.

This leads to the approch of *para-functional programming* [Hudak, 1986] where the user can by explicit annotations influence the evaluation order in functional programs without changing the correctness. The disadvantage of this approach is the fact that the program source is forced into a particular structure (a composition of higher order functions). Moreover, it would be nice if the compiler iself could *suggest* parallelization points (where the programmer could select the most attractive ones) or *explain* why certain program parts are not worth for parallelization (*semi-automatic parallelization*).

### 9.1.3 Abstract Interpretation

Another approach would is to ignore the internal structure of the program but to concentrate on its *extension* i.e. its input/output relationship. The basic idea of abstract interpretation is to introduce an *abstraction map* $\alpha : S \to A$ (where $S$ is the standard program domain and $A$ is an abstract domain) that defines some interesting runtime property of a program by mapping concrete program objects into elements of the abstract domain. E.g. we may define $\alpha : S \to \{0,1\}$ as

$$\alpha(p) = 0 \quad \Leftrightarrow \quad \texttt{p} \text{ terminates}$$
$$\alpha(p) = 1 \quad \Leftrightarrow \quad \texttt{p} \text{ runs forever}$$

where $p = \mathcal{S}(\mathtt{p})$ is the standard interpretation $\mathcal{S}$ of the program `p` (i.e. the usual meaning associated to `p`). The map $\alpha$ itself is not computable in a finite amount of time. However, we may find some *abstract interpretation* $\mathcal{A}$ that can be effectively computed and is related to $\alpha$ by the following *safety condition*

$$\alpha(\mathcal{S}(\mathtt{p})) \sqsubseteq_A \mathcal{A}(\mathtt{p})$$

where $\sqsubseteq_A$ is a partial order on the abstract domain $A$. If a compiler is then able to determine that $\mathcal{A}(\mathtt{p}) = 0$, we know that the evaluation of `p` does not terminate (while if we derive $\mathcal{A}(\mathtt{p}) = 1$ we have no information at all).

This technique is especially applied for *strictness analysis* that aims to answer the following question: how far must $x$ be known, such that a certain amount of $f(x)$ can be computed (e.g. to compute `length(l)`, only the structure of the input list `l` must be known, while for `sum(l)` also the element values are required). One idea is therefore be to compare the *strictness properties* of concrete programs to the strictness properties of functional patterns.

The main drawback of this approach is the fact that (since abstract interpretation must be *conservative* with respect to the safety condition) the delivered results may be not very accurate.

### 9.1.4 Dataflow Analysis

Another idea is to analyze the flow of data between certain characteristic points of a function and from this dataflow graph draw conclusions about the behavior of the program. This approach is sited somewhere in the middle between the intensional method of pattern matching (and may therefore more practical) and the extensional method of abstract interpretation (and may therefore deliver more accurate results). In the next subsection, we will develop this idea in more detail.

### 9.1.5 Summary

In some sense, we applied all these techniques already in the previous sections. We treated concrete functions as instances of the functional patterns, we compared new functions against already known patterns, we argued about the amount of information necessary to compute some amount of the result and we analyzed the data dependencies between different parts of the program. Now we want to systemize this process a little bit by devising some scheme that may be applied by a compiler for the automatic parallelization of list-based functional programs.

Before the analysis starts, a *simplification process* should take place where all constant expressions are folded, the applications of non-recursive functions are replaced by the corresponding function bodies and mutually recursive function definitions are transformed into directly recursive definitions (whenever this is possible). The result of this process of *partial evaluation* [Axford, 1987] is then a somewhat normalized program, namely a lattice (in most cases a partial order) of directly recursive function definitions.

## 9.2 Bottom-Up Analysis

In this subsection, we investigate the results that can be derived from analysis steps that proceed in a *bottom-up* fashion, i.e. the separate parts of a program are analyzed without taking into account the contextual information.

### 9.2.1 Type Analysis

A first rough analysis of the program can be performed on the level of the *type signatures* of functions: by comparing the nesting levels of `List` constructors (i.e. the structural complexity) of the argument and result type of a function, we may classify it as `expand` (the complexity increases), as `fold` (the complexity decreases) or as `transform` respectively `merge` (the complexity stays the same).

This rough distinction already gives some useful information, since parallelism can be typically derived from functions that generate objects of higher (or at least the same) complexity as the arguments than from functions. However, in general this straight-forward approach alone is too weak and can just serve as the basis of the following analysis steps.

### 9.2.2 Strictness Analysis

A common abstract domain for the strictness analysis of functional programs involving list structures is the *four point domain* introduced by [Wadler, 1987]:

$$\begin{array}{lcl} \alpha(l) & = & 0 \ \ldots\ l \text{ is undefined at all} \\ \alpha(l) & = & 1 \ \ldots\ l \text{ has some undefined tail} \\ \alpha(l) & = & 2 \ \ldots\ l \text{ has some undefined element} \\ \alpha(l) & = & 3 \ \ldots\ l \text{ is fully defined} \end{array}$$

By abstract interpretation, an approximation of the strictness interface of each function can be determined and from this information some conclusions about the dynamical behavior of a function may be drawn.

- If $\bar{f}(\ldots, 0, \ldots) > 0$ (where $\bar{f}$ is the abstracted version of a function $f$), $f$ does not require the argument abstracted to 0 in order to return some amount of the result. This behavior typically occurs for an extra list argument that is used for *implicit concatentation*. It also says that the function that computes this argument can run independently of and in parallel with $f$ (*quasi-horizontal parallelism*).

- If $\bar{f}(\ldots, 1, \ldots) > 0$, we must only know whether the corresponding input list is `nil` or not in order to return some amount of the result. This behavior is typical for `transform` (or `merge`) like functions that generate their results in a *stream-like* fashion regardless of the concrete element values.
- If $\bar{f}(\ldots, 2, \ldots) > 0$, then the whole backbone of the input list must be traversed in order to return the result. This behavior typically occurs if $f$ is defined tail-recursively on the structure of the input list and delivers its result in a *array-like* fashion in the base case of recursion only.
- If only $\bar{f}(\ldots, 3, \ldots) > 0$, then also some elements of the input structue must be available in order to return some amount of the result. This is typically true for folding functions where the operator `f` is strict in its arguments or for conditional list processors where the value of the current head element controls the computation.

This four point domain only delivers information about the behavior of a function with respect to the top-level `List` constructor of its arguments and of its result (i.e. the list elements themselves are considered atomic). However, it is possible to extend this domain systematically also for nested types such that more accurate information e.g. for lists of lists may be derived [Burn, 1991].

Strictness analysis is restricted in several ways: first, only an approximation of the abstraction map can be computed that must be conservative with respect to the safety condition. Second and more important, the analysis is only able to tell us *whether* an argument is required but not *when*; thus the dynamic behavior of nested program parts (vertical parallelism) cannot be adequately described.

Moreover, strictness analysis by abstract interpretation is not able to formalize the concept of *head strictness*: the behavior of a function that requires only the value of the current head of the input list to produce some amount of the result cannot be distinguished from that of a function that needs *all* list elements to return a value. There are other techniques than abstract interpretation to overcome this problem [Wadler and Hughes, 1987], but these techniques are also subject to the limitations mentioned above.

### 9.2.3 Weight Analysis

Abstract interpretation is also able to give us some information about the *weight* of different program parts by selecting the execution time as the domain of abstraction. However, in contrast to strictness domains, this abstract domain is not bounded and abstract interpretation will never reach a fix point (i.e. not terminate) for recursive function definitions.

To overcome this problem, we may bound the number of iteration steps in fixpoint computation, i.e. we assume some maximum depth for recursive function calls [Goldberg, 1988]. An extension to this idea would be to introduce a combined abstract domain that not only approximates the *time* required to evaluate some expression but also the *complexity* of the result (i.e. the length of a result list). For structually recursive functions (i.e. iterating on list arguments), more accurate results can then probably be provided.

### 9.2.4 Body Structure Analysis

The former analysis steps did not look inside the body of a function in order to draw conclusions from its concrete structure. As already mentioned, it has not much sense to attempt to match concrete programs against the idealized patterns presented in this paper, since the number of possible variations is too large. We therefore must find some abstracting filter that allows us to focus on the crucial parts of the program. For our purposes, these crucial parts are

1. **Selections** (`case` statements) that decompose input lists into substructures and determine by case distinction the further computation process.
2. **Conditionals** (`if` statements) that determine the further computation process by the result of a boolean test condition.
3. **Recursive Function Applications** that determine the "recursion pattern" of the function and are the origin of parallel process creation.

A first step of this analysis is the creation of a tree of possible evaluation pathes within a function $f$ consisting of two kinds of interior nodes:

1. **case nodes** that are labelled with the associated list expressions.
2. **if nodes** that are labelled with the boolean test condition.

Each node has a number of successor nodes that represent the expressions that are evaluated if the corresponding branch is taken. For each leaf in the tree, the structure and the dataflow in the corresponding result expression can be analyzed with respect to recursive function applications. The following cases of result expressions are possible:

1. **No recursion:** no application of $f$.
2. **Recursion:** $c(\ldots, f(\ldots), \ldots)$.
3. **Tail recursion:** $f(\ldots)$.
4. **Nested recursion:** $c(\ldots, f(\ldots, f(\ldots), \ldots), \ldots)$.
5. **Nested tail recursion:** $f(\ldots, f(\ldots), \ldots)$.

6. **Independent recursion:** $c(\ldots, f(\ldots), \ldots, f(\ldots), \ldots)$.
7. **Others:** more than two applications of $f$.

Please note, we do not demand that the actual result expression has the *pattern* depicted above but shows the corresponding *flow of data* (regardless the concrete syntactic structure).

### 9.2.5 Dataflow Analysis

For each *recursive function call* in the body of the function, the following objects should be determined:

1. The set of **case patterns** $p$ that have to be matched before the corresponding branch is taken and the function can be recursively called.
2. The set of **test conditions** $t$ that have to be evaluated before the function can be recursively called.
3. The **constructor expression** $c$ that generates the final result from the result of the recursive application.
4. The **modification expression** $m$, i.e. the largest subexpression of the constructor that does not depend on the result of the recursive function application.
5. The **successor expression** $s$ that describe how each argument of the recursive function application(s) is computed from the input arguments.

For each function argument and for each result of a recursive function application, the flow of data to $p$, $t$, $c$, $m$ and $s$ must be analyzed to determine which object is used in which part of the body. For list objects, this analysis should be performed for the head and the tail separately (if the list is decomposed by a `case` construct).

The goal is to analyze the *downward communication* structure of the function by classifiying each input list $l$ as follows:

1. **Decomposition:** Is $l$ decomposed into its head and tail or just forwarded to other functions ("no", "yes")?
2. **Conditional:** Does the result of some $p$ depend on $l$ ("no", "head only", "whole list")?
3. **Modification:** Is $l$ used in $m$ to compute the head element of the result list ("no", "head only", "whole list")?
4. **Construction:** Is $l$ used in $c$ to construct the structure of the result list ("no", "head only", "whole list")?

5. **Recursion:** Is $l$ passed to a recursive application of the current function ("no", "tail only", "whole list")?

A similar analysis (except the last of the listed items) can be performed for the result list of the recursive function applications in order to determine the structure of *upward communication*. We then are able to answer the following questions concerning the parallelization of the function body:

1. **List Usage:** How is each list argument used? Is it *traversed* in such a way that in each recursion level only the current head element is needed and the tail is always forwarded to the recursive invocation? In this case, the creation of the list may overlap with its consumption (stream parallelism). Or is the argument as a whole *forwarded* to other functions? Provided that it is not used in a conditional, the current function may then execute in parallel with the creation of the list.

2. **Conditionals:** Is the list argument used for *selecting* the branch of a conditional statement? If this is true, is the selection only based on the current list head (*head-strictness*) or does it require the whole list? In the former case, the function may already return a result if some leading elements of the list have been computed. Otherwise, the function cannot proceed execution unless the whole list and all of its elements are available (and therefore an overlapping creation of the list does not make sense at all).

3. **Construction:** Is the constructor operation *strict* in the result of the recursive function application? In this case, parallelism can be derived only from the parallel evaluation of the modification functions $m$, Otherwise there is also vertical parallelism between the different instances of $c$. Moreover, are the result lists generated in a *stream-way* fashion or (by tail-recursion) *array-like*? Are the list elements computation in the *forward* direction or is there any data dependency by upward communication that forces element computation in the *reverse* order?

Furthermore, a *weight analysis* is to be performed for the subfunctions $p$, $m$, $s$ and $c$ to determine their relative complexity. The parallelization of the function body (by starting parallel processes for the recursive function calls) basically only pays in the following cases:

1. **Data Parallelism:** The modification operation `m` is costly and the various invocations of `m` can execute in parallel.

2. **Vertical Parallelism:** The general construction operation `C` is costly and can run in parallel with the recursive function application.

3. **(Quasi-)Horizontal Parallelism:** There are two independent recursive function applications or two nested function applications that can execute in parallel.

#### 9.2.6 Results

The result of the bottom-up analysis phase consists of the following pieces of information for each function:

1. **Parallelization Strategy** An execution model is suggested for each program function that can be classified according to the previously presented criteria: sequential execution (no internal parallelism), data parallelism (parallel element computation), vertical parallelism (overlapping structure generation) or (quasi-)horizontal parallelism (divide and conquer).

2. **Argument Classification** It is predicted to which extent each recursion level requires each argument to proceed execution (assuming the suggested execution model): not at all, an initial segment of the list structure, an initial segment and the corresponding elements, the whole list structure or the whole structure and all elements.

3. **Result Classification** For each list of the result tuple, the information is delivered whether the list is generated stream-like or array-like (by tail-recursion) and in which direction the list elements are computed (left to right or right to left).

These results will (together with those of *strictness analysis* and *weight analysis*) be applied by the following phase to determine which function applications will be actually parallelized.

### 9.3 Top-Down Analysis

While in the bottom-up phase, the separate program functions were analyzed for possibilities of parallelization, in the top-down analysis phase the structure of the whole program graph (i.e. the composition of functions) shall be taken into account.

#### 9.3.1 Internal Parallelization

Which applications of list-processing functions suggested by the bottom-up phase are actually parallelized in their internal structure (i.e. parallel processes are started for their recursive function applications) and which are still executed sequentially? This is mainly a question of the relative weight of the application with respect to the rest of the program.

Starting from the top-level function application, the analysis should proceed downwards to find the outermost function that the bottom-up step suggested for internal parallelization. However, only if the granularity of the application is sufficiently large compared to the granularity of the total program, the internal structure of the function will be actually unfold into parallel processes.

In order to keep the process granularity large, in general only one level of program structures should be parallelized (the outermost recursion in the ideal case). I.e. if a function whose recursive structure is already executed in parallel calls another function, the subfunction should be itself executed sequentially. For massively parallel machines, at most two or three levels of parallelization will in general suffice to saturate the processsors with work.

### 9.3.2 Parallel Composition

Which applications of list-processing functions are executed as parallel processes in the background and which are executed by the current process? The question arises also for functions that are executed sequentially (due to the possibilities of horizontal parallelism an of stream parallelism) and is mainly influenced by the data dependencies between functions and by the way they process list arguments and generate result lists.

If a program function is a compositon of list processing functions of large granularity, the functions can be executed in parallel if they are totally independent (*horizontal parallelism*) or if the result of one application is within the consuming application just forwarded to other processes (*quasi-horizontal parallelism*).

*Vertical parallelism* between list-processing functions makes sense if both the producer and the consumer are both internally parallelized (since then the elements are produced and consumed by independent parallel processes). If this is not the case, still *stream-parallelism* is possible provided that the producer generates the list structure in a stream-like fashion (and the element values are computed from left to right) and eah level of recursion in the the consumer process only depends on the current head of the list.

### 9.3.3 User Support

Several of these strategic decisions may also be supported by *additional compile-time information* provided by the user: the number of available processors, the desired "depth" of parallelization, the length of typical input lists or the maximum depth of recursion for several functions. Also program annotations for "good" points of parallelization (or vice versa functions that should be executed sequentially) will help to improve the efficiency of the generated code.

# 10 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). By automatic analysis techniques (pattern matching, abstract interpretation, dataflow analysis), 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. By compile-time information provided by the human programmer, the decision process for several strategic questions (number of processes and their granularity) can be additionally simplified. 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

[Arvind and Nikhil, 1988]
Arvind and R. S. Nikhil.
Executing a Program on the MIT Tagged-Token Dataflow Architecture.
Computation Structures Group Memo 271, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1988.
Also in: 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.

[Axford, 1987]
T. H. Axford.
Partial Evaluation in a Functional Language.
Technical Report Document 15, Department of Computer Science, Functional Language Implementation Project, University of Birmingham, November 1987.
Also in: University of Birmingham, Internal Report CSR-87-9.

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

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

[Clark and Gregory, 1984]
Keith Clark and Steve Gregory.
PARLOG: Parallel Programming in Logic.
Research Report DOC 84/4, Department of Computing, Imperial College of Science and Technology, University of London, April 1984.
Revised June 1985.

[Foster and Taylor, 1989]
Ian Foster and Stephen Taylor.
*Strand — New Concepts in Parallel Programming.*
Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

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

[Hudak *et al.*, 1991]
Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joeseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson.
Report on the Functional Programming Language Haskell.
Research Report 1.1, Department of Computing Science, University of Glasgow, Glasgow, UK, June 1991.
Final Draft.

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

[Siegl, 1990]
Kurt Siegl.
Gröbner Bases Computation in STRAND: A Case Study for Concurrent Symbolic Computation in Logic Programming Languages.
Master's thesis, Johannes Kepler University, Linz, Austria, 1990.
Also: Technical Report 90-54, RISC-Linz, Johannes Kepler University, Linz, Austria, 1990.

[Wadler and Hughes, 1987]
Phil Wadler and R. J. M. Hughes.
Projections for Strictness Analysis.
In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 385–407, Portland, Oregon, USA, September 14–16, 1987. Springer, Berlin.

[Wadler, 1987]
Philip Wadler.
*Strictness Analysis on Non-Flat Domains (by Abstract Interpretation over Finite Domains.*
Ellis Horwood, 1987.

[Wikström, 1987]
A. Wikström.
*Functional Programming Using Standard ML.*

International Series in Computer Science. Prentice-Hall, 1987.

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