A Transformational Approach which Combines Size Inference and Program Optimization

Position Paper

Christoph A. Herrmann and Christian Lengauer

Fakultät für Mathematik und Informatik,
Universität Passau, Germany

`{herrmann,lengauer}@fmi.uni-passau.de`
http://www.fmi.uni-passau.de/cl/hdc/

Abstract. We propose a calculus for the analysis of list lengths in functional programs. In contrast to common type-based approaches, it is based on the syntactical structure of the program. To our knowledge, no other approach provides such a detailed analysis of nested lists.
The analysis of lists is preceded by a program transformation which makes sizes explicit as program values and eliminates the chain of cons operations. This permits alternative implementations of lists, e.g., by functions or arrays. The technique is being implemented in an experimental parallelizing compiler for the functional language $\mathcal{HDC}$.
We believe that analysis and parallelization work best if higher-order functions are used to compose the program from functional building blocks, so-called skeletons, instead of using unrestrained recursion. Skeletons, e.g., data-parallel combinators come with a theory of sizes and parallelization.

1 Introduction

If functional programs are to be used for high-performance computing, efficient data representations and operations must be provided. Our contribution is a calculus for the analysis of the lengths of (nested) lists and a transformation into a form which is liberated from the chain of cons-operations and which sometimes permits array implementations even if the length depends on run-time values.

A major advantage of functional programs vs. imperative programs is that dependence analysis is much easier, due to the absence of reassignments. One severe disadvantage of functional programs as of yet is that efficient, machine-oriented data structures (like the array) –absolutely necessary for high-performance computing– play a minor role in many language implementations since they do not harmonize with functional evaluation schemata (like graph reduction), which are at a higher level of abstraction.

We propose to construct programs by composition of skeletons, i.e., functional building blocks with a predefined, efficient implementation [9]. From the view of the source program, they are higher-order functions which are instantiated with problem-specific, customizing functions. We implement skeletons in an imperative language close to the machine. In the compilation of the program parts which are not skeletons, functional concepts are successively eliminated such that these parts can be linked together with the skeleton implementations. In this process, the most important step is the replacement of functional arguments by data structures of the source language [1].

Aside from instantiation of skeletons, functional arguments should be used moderately since they incur overhead and might introduce undesired dependences. Recursion should be replaced by skeletons, e.g., the recursive function `map`, which applies a function to each element of a list, can be replaced by a data-parallel implementation. The need for a size analysis arises from the use of simple inductive data structures, e.g., the list. With knowledge of its length, the list might be implemented more efficiently as an array.

Our size analysis calculates information about the sizes of lists at compile time, in terms of structural parameters, i.e., symbolic names assigned to the lengths of lists in the input. Characteristic for our approach is that the size analysis also computes a function which maps indices to elements. The result of the analysis can then be used for optimization by program transformations, e.g., intermediate lists could be eliminated, similar to deforestation or `map` distribution over composition [3]. The transformations provide the basis for a renewed size inference and subsequent optimization, in an iterative process which terminates according to criteria specified by compiler settings or directives.

Our inference and transformation rules are based on a view of lists which abstracts from the chain of elements present in many standard representations. Due to the absence of side effects, the compiler is not obliged to preserve a particular representation of data structures, i.e., a list may be eliminated, fused with some other list, represented by an array, reproduced by a function, etc. Data aggregates treated in such an abstract fashion are known as data fields [15, 24].

As far as we know, we are the first to derive compile-time information about each element of a list in terms of its position. This is possible by a symbolic representation of a function mapping indices to elements – a technique which provides the potential for a precise size analysis of nested lists and for their flat implementations [30]. Flat structures can lead to efficiency increases in memory allocation and release, access and update of elements and marshaling for communication.

Through size inference, the program can become amenable to further transformation, since compile-time information becomes visible at points where it was not visible before. With this kind of compilation, the efficiency of the generated code becomes sensitive to small changes in the program and, thus, small actions of program maintenance may have dramatic effects. Note that we aim for high performance of selected program parts, achieved with the programmer's interaction, not for a compiler which produces good code fully automatically in the average case. The novice programmer can interact by setting compiler switches and providing program annotations. The advanced (parallel) programmer can add skeleton implementations which capture computation schemata not previously known to the compiler.

Sect. 2 reviews related approaches to size inference. As a motivation for size inference, we present our experimental compiler in Sect. 3. Sect. 4 presents a transformation of the list data type which makes size expressions explicit in the program. In Sect. 5, we discuss the simplification of size expressions in a little auxiliary language, which need not –and, indeed, does not– contain lists, because size expressions have been disentangled from the list contents by the transformation in Sect. 4. Sect. 6 presents an example for which an exact treatment of the sizes of nested lists is useful: the multiplication of polynomials with coefficients represented by lists of digits. In Sect. 7, we summarize our statements and point to future work.

2 Related Work

The data field approach of Hammarlund and Lisper [15] inspired us to abstract from actual representations of aggregate data objects, in favor of minimizing dependences between data and increasing the potential for parallelism. By using an indexing function to refer to elements of an aggregate structure, many *arrangement* operations can be performed without visiting the data at all, just by modification of the indexing function: permutation, broadcast, partitioning, etc. We apply the data field approach to the parallelization of lists. As far as we know, the list is the most important aggregate data structure in functional programming, and it has a rich theory [3, 28].

As Lisper and Collard [24] have pointed out, size inference can be viewed as a form of abstract interpretation [8]. One kind of abstract information of a list is its length. The length function is a monoid homomorphism that maps from the concrete domain of lists to the abstract domain of natural numbers. The empty list is mapped to zero and the function which adds an element to a list is mapped to the successor function. Unfortunately, this nice property of the list length is only one side of the coin. The power of abstract interpretation comes from the fact that the calculation is performed solely in the abstract domain. A complication with lists is that they can contain lists as elements. Applying the abstraction at the outer nesting level incurs a loss of information about the lengths of the lists at inner nesting levels, while an abstraction at an inner level means that the outer lists remain part of the abstract domain.

We employ a representation of lists which isolates the size information while preserving all other program information. Thus, we are doing something similar to abstract interpretation by performing a static analysis of the isolated size information. For nested lists, this means first to perform a static analysis of the lengths of the lists at the outer nesting level, then to continue with the analysis of the elements of that list, and so on.

The standard construction of lists is inductive and excludes a global view of the size or a possibility to access an element by index directly. Our new representation of lists has consequences for element access beyond program analysis. In his *views* approach, Wadler [31] proposes pattern matching with constructors that do *not* form the representation of the data structure. We apply this principle to lists: the representation is subject to optimization by the compiler, while the standard list constructors are still available for user programs. In addition, a new list constructor is introduced internally, which permits symbolic pattern matching of a list against its size and its indexing function. We took the idea of such non-standard list constructors from cons-snoc lists and distributable homomorphisms [13] and powerlists [26]. In contrast to them, our approach strictly separates length and content of a list, with the need to add auxiliary functions to the program, mapping list indices to elements. Later in the compilation process, inlining of these functions can improve efficiency.

Our approach differs significantly from others in three aspects: (1) the impact of the success of the analysis and the way it is integrated into a compilation, (2) the role of types for size inference and (3) the restrictions of the source language:

1. Our size analysis is an optional part of the compilation. Its success –more precisely, the degree of its success– determines the efficiency of the target code but cannot lead to a failure of the compilation or a rejection of the program.
   Size inference or size checking appears in other approaches at the front end of a compilation process, even in transformational approaches [12]. Our size analysis is located in the middle of the back end of the compilation, inside an optimization loop. The analysis and subsequent program transformations are performed in a cycle and, thus, functions are analyzed that never existed in the source program.
2. Other researchers base size analysis, be it inference or checking, on types.
   (a) Some groups draw a strong connection between types and sizes. Bellè and Moggi [2] apply size inference in an intermediate language with a two-level type system that distinguishes between compile-time and run-time values [27]. Xi and Pfenning [32] use dependent types and perform type checking modulo constraint satisfaction. Singleton types enable the generation of lists of a particular size, dependent on integer values. Jay and Sekanina [23] describe type checking in a language VEC on vectors, which distinguishes between so-called *shapely* types and *non-shapely* types. They distinguish two kinds of conditionals. The shapely conditional, which requires the condition to be of shapely type, is used to analyze recursive size functions but cannot deal with arbitrary program values. The non-shapely conditional can deal with all program values but cannot be used to define sizes. A surprisingly large class of programs can be handled with this approach: all usual array indexing operations and linear algebra. Hofmann [21] uses a linear type system to control the size of lists and permit an in-place update which is especially useful for sorting. Chin and Khoo [6] infer the sizes of recursive functions by a fixed-point calculation.

(b) Others perform type inference first, to keep it decidable, and then infer sizes based on the type information. Hughes, Pareto and Sabry [22] employ three separate, successive steps: (1) Hindley-Milner inference, (2) derivation of the size information and (3) constraint solving. Loidl and Hammond [25] followed this approach. Our initial attempts were similar [17], but we recognized that the treatment of nested lists in the type information leads to a formalism which is difficult to apply. Now, we are using types only to count the levels of nesting of the list, but do not tag type information with size information. As far as we know, the other groups are considering nested lists of rectangular shape only, i.e., those isomorphic to multi-dimensional arrays. This simplifies their treatment in the type system significantly. Fradet and Mallet [12] use predicates to restrict operations to a subset of this rectangular shape.

3. We do not impose restrictions on the source language to perform size analysis. As a necessary consequence, arbitrary size expressions may enter the analysis and, thus, conditions on sizes may be undecidable. Since our type inference happens in an earlier compiler phase and is completely unrelated to our static size analysis, the size analysis may fail but the program may still be typable.

   In all other static approaches which we are aware of size inference is decidable. Chin and Khoo [6], Fradet and Mallet [12] and Xi and Pfenning [32] use linear inequalities for constraints, i.e., Presburger arithmetic. Bellè and Moggi [2], Jay and Sekanina [23] and Hofmann [21] achieve the decidability through their type system.

   Limitation to Presburger arithmetic already rules out the following simple function f which is likely to appear in an N-body computation: f takes a list of size m as input and produces a list that contains all element pairs and whose size expression m^2 is not permitted in a Presburger formula.

   Although formulas of number theory are, in general, undecidable, there is a good chance that one can solve more formulas than just those of Presburger arithmetic. Our approach is based on an extensible library of formula patterns and their simplifications, which are being consulted in size inference.

3 The $\mathcal{HDC}$ Compiler

Functional languages are well suited for program parallelization because of the simplicity of program analysis and transformation compared to imperative languages. However, if functional languages are to be competitive, their compilers must produce similar target code, without overhead for memory management or auxiliary data structures. This has motivated us to develop our own compiler for a functional language, with a strong focus on the code it produces. Skeletons are a suitable vehicle for achieving high efficiency [19].

We name our source language $\mathcal{HDC}$, for $\mathcal{H}$igher-order $\mathcal{D}$ivide-and-$\mathcal{C}$onquer, which reflects our belief in the productivity that higher-order functions lend to programming. We have focussed on divide-and-conquer because of its high potential for parallelism. Efficient parallel implementations of divide-and-conquer skeletons have been added to the compiler and have been used to express algorithms like Karatsuba's polynomial multiplication and the n-queens problem [20].

The syntax of $\mathcal{HDC}$ is a subset of Haskell [16], since we found language constructs like the list comprehension superior concerning notational convenience and style. In contrast to Haskell, the semantics of $\mathcal{HDC}$ is strict. This is to guarantee that the space-time mapping –that is, the assignment of operations to time and processors made at compile time– is respected in the actual execution.

Briefly, the $\mathcal{HDC}$ compiler [18] consists of the following parts:

1. front end: scanning, parsing, desugaring, λ-lifting, type inference/checking
2. back end, pre-optimization part: monomorphization, elimination of functional arguments, elimination of mutual recursion, case-elimination, generation of a directed acyclic program graph, tuple elimination
3. back end, optimization cycle:
   (a) inlining (unfolding of function applications)
   (b) rule-based optimizations by transformation (Sect. 4)
   (c) size inference (Sect. 5)
4. back end, post-optimization part: abstract code generation, space-time mapping (parallelization), code generation, skeleton instantiation

In order to increase code efficiency, size inference and code optimizing transformations are performed alternatingly in several iterations. The size information can be used to control program transformations in the next iteration. Additionally, the size information is useful in the parallelization and for memory assignment in the code generation.

Due to the complexity of the task, the implementation of size inference is still at an early stage. Thus, the experimental results available to-date [18, 20] do not reflect the impact of the methods presented here.

4 A List Transformation for Size Inference

This section is about the transformation which makes size information explicit for the later size analysis. In the source, lists are represented by the constructor nil ([ ]) for the empty list and the constructor cons (:) for appending a new element at the front of a list. The dependences introduced by cons rule out a constant-time access to all list elements and complicate the analysis. After the transformation, each list is represented by a pair of its size and a function mapping indices to elements. The transformation itself appears straightforward with the calculus presented in Sect. 4.2. The difficulty is simplifying the occurring expressions of the result to a closed form – in general, this is undecidable.

4.1 A New Representation for Lists

To maintain the list xs as a data object, its length ($\#xs$) and its indexing function ($\lambda i.xs\texttt{!!}i$) are not kept separate but are combined by a new constructor which we denote by Γ. In the new representation, ($\Gamma\ f\ n$) denotes a list of length n and its i-th element ($i \geq 0$) has the value $f(i)$, e.g., the indexing function f for the list $[0, 1, 4, 9, 16]$ is given by $f(i) = i^2$. The indexing functions of lists appearing as formal parameters and at the left side of let-definitions are referred to by fresh variables. For other lists, like the arithmetic sequence $[m\,\texttt{..}\,n]$, which contains all integers from m to n, a new auxiliary function is generated and its name is used in the Γ-expression. The indexing functions are inspected and simplified during the analysis. In the next iteration of the optimization cycle, they may disappear again due to inlining. We consider two distinct implementations of Γ-expressions: (1) preferably using an explicit indexing function and (2) alternatively using an array in the case that run-time values are to be stored.

In contrast to abstract interpretation, our transformation makes the information we want to reason about –the length– explicit without incurring a loss of information. To emphasize this fact, we call this change of representation a *change of basis*. We use the notion *basis* for a set of constructors that constitute a data type.

Lemma 1 (Existence of an index/length basis). *All finite lists can be expressed in a basis which makes their length and their indexing function explicit.*

Proof. By an isomorphism of types [11] induced by the functions `to-`Γ and `from-`Γ defined below. The domain of `to-`Γ is the set of all finite lists, the codomain is restricted to the image of `to-`Γ and is taken for the domain of `from-`Γ. `map` applies f to each element of this list. Γ can be defined as an algebraic data type (data).

data $\Gamma\alpha$ = $\Gamma\ (\mathbb{N} \rightarrow \alpha)\ \mathbb{N}$

`to-`$\Gamma \in [\alpha] \rightarrow \Gamma\alpha$
`to-`$\Gamma\ xs$ = $\Gamma\ (\lambda i.xs\texttt{!!}i)\ (\#xs)$

`from-`$\Gamma \in \Gamma\alpha \rightarrow [\alpha]$
`from-`$\Gamma\ (\Gamma\ f\ n)$ = `map` $f\ [0\,\texttt{..}\,n-1]$

□

Due to this isomorphism, we identify the types $\Gamma\alpha$ and $[\alpha]$ and enable the compiler to apply `to-`Γ and `from-`Γ where an adaptation is required. The notation we use is adapted from the language Haskell [16].

4.2 Rewriting in the Γ-Calculus

The Γ-calculus is a set of rules that can be used for converting the standard list representation to Γ-form. Tab. 1 explains the notation used in the calculus. We split the rules into three classes. The top of Fig. 1 gives a complete specification of the semantics of Γ. In the middle part of the figure, we define the basic list functions in terms of Γ. Since pattern matching has been eliminated in an earlier compiler phase, we rely on a predicate for the empty list and two functions for selecting head and tail of a list, in addition to nil and cons. The rules for these basic functions are consistent with the definition based on the representation with nil and cons.

$a \downarrow b \;/\; a \uparrow b$	minimum/maximum of a and b
$a \updownarrow_{low}^{high}$	$= \; (a \uparrow low) \downarrow high$
$\otimes$	for an arbitrary binary operator of type $\alpha \to \beta \to \alpha$
${}^{e}\bigotimes_{i=low}^{high} x_i$	$= \begin{cases} ((e \otimes x_{low}) \otimes \ldots) \otimes x_{high}, & \text{if } low \leq high \\ e & \text{otherwise} \end{cases}$
$e[x{:=}v]$	substitute every free occurrence of x in e by v
$\mu \; k \; x_i$	the highest j, for which $\sum_{i=0}^{j-1} x_i$ $(x_i \in \mathbb{N})$ does not exceed k

Table 1. Notation used in the calculus

The rules in the lower part of Fig. 1 are derived from the basic list functions and simplified. Our strategy is to use as many rules as possible to accelerate the simplification of size information. Every list function, for which a rule is not stated, must be analyzed itself.

The following lemma states that these rules can be used to construct a terminating rewrite system. (We do not need confluence, since we do not compare the simplified program parts.)

Lemma 2 (Termination of the reduced rewrite system). *In the Γ-calculus without rule* **intr-Γ***, rewriting terminates with an expression which contains neither the constructors nil and cons nor any list function on the left side of a rule.*

Proof sketch. The number of occurrences of nil, cons and list functions in an expression is finite. Each application of a rule strictly decreases this number by at least one. □

4.3 The List Transformation Algorithm

In many circumstances, the change of basis delivers a form of the function which expresses the lengths of the result lists in terms of the lengths of the argument lists. The difficulty is that this reduced form will likely inherit the recursion of the original. In Sect. 5, we tackle recursion elimination and other simplifications of size expressions in a little language. This language has only the value domains

intr-Γ	$\{xs$ is a list$\}$	$xs \overset{\text{fresh } i}{\longrightarrow} \Gamma\ (\lambda i.xs\,\texttt{!!}\,i)\ (\#xs)$
elim-Γ.0		$\#\ (\Gamma\ \_\ n) \longrightarrow n$
elim-Γ.1		$(\Gamma\ f\ n)\,\texttt{!!}\,i \longrightarrow \underline{\texttt{if}}\ 0 \leq i < n\ \underline{\texttt{then}}\ f\ i\ \underline{\texttt{else}}\ \bot$
null-Γ		$\texttt{null}\ (\Gamma\ \_\ n) \longrightarrow n = 0$
nil-Γ		$[\,] \longrightarrow \Gamma\ (\texttt{const}\ \bot)\ 0$
cons-Γ		$x : \Gamma\ f\ n \overset{\text{fresh } i}{\longrightarrow} \Gamma\ (\lambda i.\underline{\texttt{if}}\ i = 0\ \underline{\texttt{then}}\ x\ \underline{\texttt{else}}\ f\ (i-1))\ (n+1)$
head-Γ		$\texttt{head}\ (\Gamma\ f\ n) \longrightarrow \underline{\texttt{if}}\ n > 0\ \underline{\texttt{then}}\ f\ 0\ \underline{\texttt{else}}\ \bot$
tail-Γ		$\texttt{tail}\ (\Gamma\ f\ n) \longrightarrow \underline{\texttt{if}}\ n > 0\ \underline{\texttt{then}}\ \Gamma\ (f \circ (+1))\ (n-1)\ \underline{\texttt{else}}\ \bot$
sequence-Γ		$[a\,..\,b] \longrightarrow \Gamma\ (+a)\ ((b-a+1) \uparrow 0)$
take-Γ		$\texttt{take}\ k\ (\Gamma\ f\ n) \longrightarrow \Gamma\ f\ (k \updownarrow_0^n)$
drop-Γ		$\texttt{drop}\ k\ (\Gamma\ f\ n) \longrightarrow \Gamma\ (f \circ (+(k \updownarrow_0^n)))\ (n - (k \updownarrow_0^n))$
map-Γ		$\texttt{map}\ g\ (\Gamma\ f\ n) \longrightarrow \Gamma\ (g \circ f)\ n$
foldl-Γ		$\texttt{foldl}\ \otimes\ e\ (\Gamma\ f\ n) \overset{\text{fresh } i}{\longrightarrow} {}^{e}\bigotimes_{i=0}^{n-1} (f\ i)$
scanl-Γ		$\texttt{scanl}\ \otimes\ e\ (\Gamma\ f\ n)$ $\overset{\text{fresh } i,j}{\longrightarrow} \Gamma\ (\lambda j.\ {}^{e}\bigotimes_{i=0}^{j-1} (f\ i))\ (n+1)$
append-Γ		$\Gamma\ f\ m +\!\!+\ \Gamma\ g\ n$ $\overset{\text{fresh } i}{\longrightarrow} \Gamma\ (\lambda i.\underline{\texttt{if}}\ i < m\ \underline{\texttt{then}}\ f\ i\ \underline{\texttt{else}}\ g\ (i-m))\ (m+n)$
concat-Γ		$\texttt{concat}\ (\Gamma\ (\lambda i.(\Gamma\ (\lambda j.e_{i,j})\ n_i))\ m)$ $\overset{\text{fresh } k}{\longrightarrow} \Gamma\ (\lambda k.((e_{i,j}[i := \mu\ k\ n_i])[j := k - \sum_{i=0}^{\mu\ k\ n_i - 1} n_i]))\ (\sum_{i=0}^{m-1} n_i)$

Fig. 1. Rewrite rules

of numbers and Booleans, but contains symbolic reduction operators, e.g., summation. The symbolic calculation is necessary since the lengths of the argument lists are unknown at compile time and are, thus, represented by variables.

Our algorithm works on the syntax tree of the function. From an algorithmic point of view, the change of basis simplifies our transformation, since each list (in Γ-form) carries its (symbolic) length information along. If lengths were made part of the type information, the correctness of the transformation could not be established solely by equational reasoning about the functional expressions. Also, nested lists can be treated precisely, since the length of an inner list can be expressed in terms of its index in the outer list. Algorithm LISTSIMP (Fig. 2) performs a complete change of basis on the lists in the expression given to it.

```
INPUT: expression $e$ and a set of constraints
OUTPUT: expression semantically equivalent to $e$ which does not
contain list operations in the standard basis

if $e$ is a constant or variable
then if $e$ is not of a list type
        then return $e$
        else substitute every occurrence of $e$ by $(\Gamma\ f\ n)$
        where $f$ and $n$ are fresh names
else ($e$ is a compound expression):
  1. apply LISTSIMP to each component of $e$; the result is called $e'$
  2. perform simplifications in the size language of all arithmetic ex-
     pressions in $e'$, yielding $e''$
  3. eliminate the standard list constructors and functions from the
     current node of the syntax tree by applying the rule of the $\Gamma$-
     calculus that matches, obtaining $e'''$
  4. if $e'''$ is not of a list type then return $e'''$
     else ($e'''$ is a list, represented by, say $(\Gamma\ h\ m)$):
     (a) apply LISTSIMP to the expression $m$, getting $m'$
     (b) apply LISTSIMP to the expression $h$ using knowledge of $m'$,
         yielding $h'$
     (c) return $(\Gamma\ h'\ m')$
```

Fig. 2. Algorithm LISTSIMP

We demonstrate the algorithm on a tiny, non-recursive function. Function `rotate` performs a cyclic shift of the elements of a list. Application areas are hardware description/simulation or convolution.

The beauty of lists for this purpose is obvious: to rotate the first eight items of a list *xs*, we just write: `rotate` (`take` 8 *xs*) ++ `drop` 8 *xs*.

1. The initial function is as follows:

   `rotate` xs = <u>`if`</u> $\#xs < 2$ <u>`then`</u> xs <u>`else`</u> `tail` xs ++ [`head` xs]

Note that a straightforward compilation of this function would produce nasty code. On the one hand, the expression `tail` xs cannot be shared because [`head` xs] has been appended at the end. On the other hand, it cannot be updated in-place, although it is not shared.

2. According to the algorithm, each occurrence of the list variable xs is replaced by $(\Gamma\ f\ n)$, where f and n are fresh variables:

$$\begin{aligned}\texttt{rotate}\ (\Gamma\ f\ n) = \underline{\texttt{if}}\ \#(\Gamma\ f\ n) < 2\ &\underline{\texttt{then}}\ \Gamma\ f\ n\\ &\underline{\texttt{else}}\ \texttt{tail}\ (\Gamma\ f\ n) +\!\!+ [\texttt{head}\ (\Gamma\ f\ n)]\end{aligned}$$

3. Application of the rules for #, `tail` and `head`:

$$\begin{aligned}\texttt{rotate}\ (\Gamma\ f\ n) = \underline{\texttt{if}}\ n{<}2\ &\underline{\texttt{then}}\ \Gamma\ f\ n\\ &\underline{\texttt{else}}\ \Gamma\ (\lambda i.f\,(i{+}1))\ (n{-}1) +\!\!+ [f\ 0]\end{aligned}$$

4. Application of the rules for nil and cons to $[f\ 0]$:

$$\begin{aligned}\texttt{rotate}\ (\Gamma\ f\ n) = \underline{\texttt{if}}\ n{<}2\ &\underline{\texttt{then}}\ \Gamma\ f\ n\\ &\underline{\texttt{else}}\quad \Gamma\ (\lambda i.f\,(i{+}1))\ (n{-}1)\\ &\qquad +\!\!+\ \Gamma\ (\lambda i.\underline{\texttt{if}}\ i = 0\ \underline{\texttt{then}}\ f\,0\ \underline{\texttt{else}}\ \bot)\ 1\end{aligned}$$

5. Application of the append rule:

$$\begin{aligned}\texttt{rotate}\ (\Gamma\ f\ n) = \underline{\texttt{if}}\ n{<}2\ &\underline{\texttt{then}}\ \Gamma\ f\ n\\ &\underline{\texttt{else}}\ \Gamma\ (\lambda i.\underline{\texttt{if}}\ i{<}n{-}1\ \underline{\texttt{then}}\ f\,(i{+}1)\ \underline{\texttt{else}}\ f\,0)\ n\end{aligned}$$

6. Simplification of the conditional, using information about the length:

$$\texttt{rotate}\ (\Gamma\ f\ n) = \Gamma\ (\lambda i.f\,((i{+}1)\ \mathrm{mod}\ n))\ n$$

All rule applications aside from the simplification at the end are straightforward according to the rules. The success of the simplification enables further possibilities, e.g., an optimization of a sequence of k rotations, given by the following equality:

$$\texttt{rotate}^k\ (\Gamma\ f\ n) = \Gamma\ (\lambda i.f\,((i{+}k)\ \mathrm{mod}\ n))\ n$$

5 Simplification of Size Expressions

In the previous section, we have decomposed list data objects into two independent components: indexing function and length – both symbolic arithmetic expressions. Further simplifications need not resort to the list data type anymore.

The process of size inference abstracts temporarily from the program representation to focus on mathematical issues. Our intention is to handle constraint solving, simplification, etc. in a separate package which need not know anything about the syntax or semantics of the functional programming language. In this package, we use a small, first-order functional language, the *size* language. It consists of a set of (possibly recursive) function definitions and a single expression which defines a size dependent on symbolic names and which can use the set of functions.

The size language still needs to contain recursion. E.g., here is the size function obtained from a recursive reverse function:

$$\texttt{reverseSize}\, n \;=\; \underline{\texttt{if}}\, n = 0\, \underline{\texttt{then}}\, n\, \underline{\texttt{else}}\, \texttt{reverseSize}\,(n-1) + 1$$

Simplification must solve this recursion. We will discuss that in Sect. 5.3.

5.1 The Syntax of Size Expressions

Atomic size expressions are constants and variables. Size expressions can be composed by arithmetic operators, conditionals, reduction operators (e.g., summation) and applications of size functions. Variables are used to represent unknown values and for indexing elements in a reduction.

Structural parameters are those unknowns which refer to input values of the function to be analyzed. Especially useful input values are list lengths and natural numbers which are decremented in recursive calls. However, the compiler may not always be able to decide which parameters are meant to be used as structural parameters. The user can point the compiler to a useful parameter –say n– for the problem size or depth of recursion by "branding" its name in the program: $n\bullet$. We believe that this kind of annotation is easier to use than annotations of the type language with size expressions.

The size information, expressed in terms of structural parameters, is derived by following the data flow of the function [4]. Our choice of a referentially transparent language enables a local analysis of each function. Where a function f is applied to an argument x, the size information of the result can often be computed by an application of the size function of f to the size information of x. We prefer to encode all functional closures by first-order data structures of the source language. Then, x will never be a functional closure.

In the calculation of sizes, rational numbers can appear as intermediate values. Exact rational arithmetic guarantees that no approximation errors will produce an incorrect integral size. Integers and natural numbers are treated as subtypes of the rational numbers. The integrality of decision variables can be enforced by subtype declarations. Boolean values are used for constraints.

We present the abstract syntax of our size language in Fig. 3. Since we are working with syntax trees only and abstract from parenthesization, punctuation, etc. of a potential source language, we use algebraic data type definitions instead of BNF rules. Like BNF, these algebraic data types can be used to define context-free expressions and, in addition, constitute a set of patterns to be used in transformations.

- A size program `P` consists of a list of function definitions (`F`) and an expression `S` to be evaluated.
- In a function definition (*name*,(*as*,*rs*)) of type `F`, *name* is the name of the function, *as* is a list of its parameter names and *rs* a tuple of result sizes. Functions can be defined recursively.

```
type P = ([F],S)
type F = (Id,([Id],[S]))
type Id = String
data T = TBool | TNat | TInt | TRat
data S = Num Rational | SV Id T
       | BTrue | BFalse
       | S:+:S | S:-:S | S:*:S | S:/:S | S:^:S
       | Floor S | Ceil S | Frac S
       | Abs S | Sgn S | Min S S | Max S S
       | S:=:S | S:<:S | S:<=:S
       | IsRat S | IsInt S | IsNat S
       | Not S | S:&:S | S:|:S | S:<=>:S
       | Case [(S,S)] S
       | Let (Id,S) S
       | Apply Id [S] [Id] S
       | Reduce (ROp,Id,S,S,S) S
       | Recur [[S]] S [S]
data ROp = Sum | Prod | Minimum | Maximum | And | Or
```

Fig. 3. The size language

- `Id` represents identifiers.
- `T` is a collection of types assigned to variables: `TBool` (Booleans), `TNat` (natural numbers), `TInt` (integers) and `TRat` (rational numbers). There is the usual inclusion relation between the number types which allows coercing in evaluation. Thus, there need not be a specific integral division. The type information is used by solvers as constraint information.
- `S` is the type of syntax trees for size expressions. Each alternative on the right side corresponds to a particular kind of node, named by a constructor. There are two kinds of constructors: infix constructors are denoted with surrounding colons (e.g., `:+:`), the other constructors are prefix constructors (e.g., `Floor`). The parts of an alternative aside from the constructor either contain subtrees (`S`) or attributes (e.g., `ROp`).
- `ROp` describes the set of reduction operators, i.e., accumulated applications of a binary associative and commutative operator.

5.2 Semantics

The meaning of size expressions is defined by the following denotation, where $\mathcal{I}[\![\, exp \,]\!]$ is the interpretation of expression *exp*.

- $\mathcal{I}[\![\, \texttt{Num}\; r \,]\!] = r$: a number, represented by an exact rational number. Due to the number type inclusion, it can also carry a natural or an integer.
- (`SV` *name t*) represents the variable *name* of type *t*. The value of a variable may be used as a value of a superset but, in constraint solving, the obtained result must match the type.

- $\mathcal{I}[\![\,\texttt{BTrue}\,]\!] = \text{True}$ and $\mathcal{I}[\![\,\texttt{BFalse}\,]\!] = \text{False}$: the boolean constants.
- $\mathcal{I}[\![\, a\texttt{:}\circledast\texttt{:}b \,]\!] = \mathcal{I}[\![\, a \,]\!] \circledast \mathcal{I}[\![\, b \,]\!]$ for each binary operator $\circledast$.
- $\mathcal{I}[\![\,\texttt{Floor}\; x \,]\!] = \lfloor \mathcal{I}[\![\, x \,]\!] \rfloor$, $\mathcal{I}[\![\,\texttt{Ceil}\; x \,]\!] = \lceil \mathcal{I}[\![\, x \,]\!] \rceil$,
  $\mathcal{I}[\![\,\texttt{Frac}\; x \,]\!] = \mathcal{I}[\![\, x \,]\!] - \lfloor \mathcal{I}[\![\, x \,]\!] \rfloor$, $\mathcal{I}[\![\,\texttt{Abs}\; x \,]\!] = |\mathcal{I}[\![\, x \,]\!]|$,
  $\mathcal{I}[\![\,\texttt{Sgn}\; x \,]\!] = \text{signum}\,(\mathcal{I}[\![\, x \,]\!])$
- $\mathcal{I}[\![\,\texttt{Min}\; x\; y \,]\!] = \mathcal{I}[\![\, x \,]\!] \downarrow \mathcal{I}[\![\, y \,]\!]$, $\mathcal{I}[\![\,\texttt{Max}\; x\; y \,]\!] = \mathcal{I}[\![\, x \,]\!] \uparrow \mathcal{I}[\![\, y \,]\!]$
- $\mathcal{I}[\![\,\texttt{IsRat}\; x \,]\!] = (\mathcal{I}[\![\, x \,]\!] \in \mathbb{Q})$, $\mathcal{I}[\![\,\texttt{IsInt}\; x \,]\!] = (\mathcal{I}[\![\, x \,]\!] \in \mathbb{Z})$,
  $\mathcal{I}[\![\,\texttt{IsNat}\; x \,]\!] = (\mathcal{I}[\![\, x \,]\!] \in \mathbb{N})$
- $\mathcal{I}[\![\,\texttt{Not}\; x \,]\!] = \neg(\mathcal{I}[\![\, x \,]\!])$, $\mathcal{I}[\![\, a\texttt{:\&:}b \,]\!] = \mathcal{I}[\![\, a \,]\!] \wedge \mathcal{I}[\![\, b \,]\!]$,
  $\mathcal{I}[\![\, a\texttt{:|:}b \,]\!] = \mathcal{I}[\![\, a \,]\!] \vee \mathcal{I}[\![\, b \,]\!]$, $\mathcal{I}[\![\, a\texttt{:<=>:}b \,]\!] = \mathcal{I}[\![\, a \,]\!] \Leftrightarrow \mathcal{I}[\![\, b \,]\!]$
- $\mathcal{I}[\![\,\texttt{Case}\;[(c_0,v_0),...,(c_n,v_n)]\; v_{n+1} \,]\!] = \mathcal{I}[\![\, v_j \,]\!]$, where j is smallest such that $\mathcal{I}[\![\, c_j \,]\!] = \text{True}$, with $\mathcal{I}[\![\, c_{n+1} \,]\!] = \text{True}$ by default.
- $\mathcal{I}[\![\,\texttt{Let}\;(x,v)\; e \,]\!] = \mathcal{I}[\![\,(e[x := v])\,]\!]$. The purpose of an auxiliary definition (`Let`) is to exploit common subexpressions.
- The semantics of (`Apply` f $[e_0,...,e_n]$ $[v_0,...,v_m]$ *exp*) is that the size function f is applied to the size expressions e_0 to e_n. f returns a tuple of size expressions which are bound to the variables v_0 to v_m. Then, the expression *exp*, defined in terms of these variables, is delivered.
- $\mathcal{I}[\![\,\texttt{Reduce}\;(\oplus, i, \mathit{low}, \mathit{high}, \mathit{cond}_i)\; \mathit{elem}_i \,]\!] = \bigoplus_{i \in I} \mathit{elem}_i$:
  a reduction with a commutative and associative binary operator $\oplus$, where $I = \{i \in \mathbb{Z} \mid \mathcal{I}[\![\, \mathit{low} \,]\!] \leq i \leq \mathcal{I}[\![\, \mathit{high} \,]\!] \wedge \mathcal{I}[\![\, \mathit{cond}_i \,]\!] = \text{True}\}$.
- $\mathcal{I}[\![\,\texttt{Recur}\; \hat{A}\; \hat{n}\; \hat{e} \,]\!] = \pi_0 A^n e$, where A is an $m \times m$ matrix, $n \in \mathbb{N}$ and e an m-column vector. $A = \mathcal{I}[\![\, \hat{A} \,]\!]$, $n = \mathcal{I}[\![\, \hat{n} \,]\!]$ and $e = \mathcal{I}[\![\, \hat{e} \,]\!]$. `Recur` expressions provide a closed form for some recurrences without using roots. E.g., the Fibonacci number n can be expressed by $fib(n) = \left(\begin{smallmatrix}1 & 0\end{smallmatrix}\right) \left(\begin{smallmatrix}1 & 1\\ 1 & 0\end{smallmatrix}\right)^n \left(\begin{smallmatrix}0\\ 1\end{smallmatrix}\right)$.

5.3 Simplification Heuristics

After the transformation of a recursive list function into Γ-form, length expressions may still be expressed recursively. Using knowledge about frequent decomposition patterns, one can provide a heuristic procedure to find closed forms.

Probably the patterns most often used are the decomposition of a list into (1) head and tail and (2) the left part and the right part [13, 26]. We discuss briefly head/tail decomposition here. If we are lucky, we obtain a recursive size function and its closed form of the following kind, where $a \in \mathbb{N}$ and $b \in \mathbb{N} \to \mathbb{N}$:

$$s(n) \;=\; \left\{ \begin{array}{cl} a & ,\;\; \text{if } n = 0 \\ s(n-1) + b(n) & ,\;\; \text{otherwise} \end{array} \right\} = a + \sum_{i=1}^{n} b(i)$$

If b is a polynomial or another simple kind of function, we can eliminate the summation operator [14]. E.g., if the size function originates from flattening a triangular matrix, we have $a = 0$ and $b(n) = n$. In this case, we obtain:

$$s(n) \;=\; 0 + \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

We advocate the use of an extensible library of patterns. Unfortunately, we cannot hope to find the pattern by a *syntactic* match. E.g., instead of the expression $s(n-1)+n$, we may encounter the expression $\lfloor n/2 \rfloor + s(n-1) + \lceil n/2 \rceil$ which is equal. Thus, we apply the following procedure:

1. Select a pattern which appears appropriate since it is known to be useful for the operators that appear in the expression. E.g., polynomials can be appropriate for expressions that contain only addition and multiplication.
2. Interpolate the expression with the pattern, obtaining values for the parameters of the pattern.
3. Run many tests with the instantiated pattern, to exclude a non-fitting pattern as quickly as possible.
4. Verify symbolically that the instantiated pattern equals the expression.
5. Simplify the pattern, exploiting properties gained by specialization.

We do not advocate interpolating the recursive function as a whole because a successful match will be very unlikely, even with a high number of attempts. Instead, we are looking for patterns for parts of the recursive function, which are (1) the condition that indicates the recursive case, (2) the value in the non-recursive case, (3) the expression which modifies the arguments for the recursive calls and (4) the expression which combines the result of the recursive calls. Then, we apply a recurrence elimination function according to the ensemble of patterns we obtained. A promising approach is to search for the power series of the generating function of the recursion [14]. The n-th coefficient of the power series carries the value of the recursive function for the input n. Chin and Khoo [5] developed a tupling method to reduce, in some cases, recursion in multiple variables to recursion in a single variable.

6 Example: Nested Lists in Multiple Precision Arithmetic

A major difference between our approach and those of others, e.g., [32], is that our size information can refer to the particular position in a surrounding data structure. Polynomial multiplication with multiple precision arithmetic makes use of this; here the bitsize of a coefficient in the result depends on its position.

6.1 Types and Representation

In order to make the maximal amount of information statically derivable, our programs resemble specifications of abstract digital designs [7, 10, 29]. The basic arithmetic functions –which are not presented here– are producing output lists that depend statically on the size of input lists. E.g., a function which adds two numbers delivers a sum whose size is the maximum of both inputs; a potential carry overflow is delivered in a separate component.

Each number is represented by a list of bits. Element i of each number represents the factor ($\in \{0,1\}$) of 2^i. We define the type `Polynomial`, whose values are polynomials in X, represented by their list of coefficients. Coefficients are themselves numbers. Element i of the polynomial represents the coefficient of X^i.

```
polymul ∈ Polynomial → Polynomial → Polynomial
polymul xss yss =
  let m = # xss
      n = # yss
  in [ let low = 0↑(k−n+1)
           high = (m−1)↓k
       in sumN [ mul (xss!!i) (yss!!(k−i)) | i ← [low..high] ]
     | k ← [0..m+n−2] ]
```

```
polymul $(\Gamma\ (\lambda i.\Gamma\ (\lambda j.f_{i,j})\ p_i)\ m)\ (\Gamma\ (\lambda i.\Gamma\ (\lambda j.g_{i,j})\ q_i)\ n)$
  $= \Gamma\ (\lambda k.$ let low       = $0\uparrow(k-n+1)$
                   high      = $(m-1)\downarrow k$
                   r         = $high-low+$Reduce (Maximum, $i, 0, high-low,$ BTrue)
                                            $(p_{(low+i)} + q_{(k-(low+i))})$
                   $(\Gamma\ h\ r)$ = sumN $(\Gamma\ (\lambda i.$ mul $(\Gamma\ (\lambda j.f_{i,j})\ p_i)\ (\Gamma\ (\lambda j.g_{k-i,j})\ q_{k-i}))$
                                        $(high-low+1))$
              in $(\Gamma\ h\ r))$
        $(m+n-1)$
```

Fig. 4. Transformation of `polymul` into Γ-form

6.2 The Source Function

The upper part of Fig. 4 shows the definition of function `polymul`. The application of `sumN` sums, for each coefficient k, the products of the coefficients of the two polynomials *xss* and *yss*. *low* and *high* are the index bounds of the coefficients of *xss*, in dependence of k. We define m as the length of *xss* and n as the length of *yss*. The degree of the product polynomial is the sum of the degree $m-1$ of *xss* and the degree $n-1$ of *yss*. Thus, it has $m+n-1$ coefficients, for X^0 to X^{m+n-2}. We use a Haskell [16] list comprehension to express this. A list comprehension has –in our case– the form [exp_i | $i \leftarrow$ [*lowbound*..*highbound*]] where the index variable i is taken from the integer range [*lowbound*..*highbound*] and exp_i denotes the element of the list associated with index i. Note that list comprehensions can be desugared in an early compiler phase; our compiler performs a desugaring into the form `map` $(\lambda i.exp_i)$ [*lowbound*..*highbound*].

6.3 Transformation into Γ-Form

The result of the transformation is shown in the lower part of Fig. 4. In Γ-form, we use m for the length of *xss* and n for the length of *yss*. The elements of *xss* and *yss* are expressed in terms of their position. *xss*!!i is itself a list, in Γ-form: $(\Gamma\ (\lambda j.f_{i,j})\ p_i)$. Here, p_i is the length of *xss*!!i, and $f_{i,j}$ its element with index j. The representation of *yss*!!i is analogous, with q instead of p and g instead of f. The analysis should infer a simplified Γ-expression for the body of `polymul` with respect to the following application, where *xss* and *yss* have been replaced by their Γ-form, as described above:

$$\texttt{polymul}\ \ (\Gamma\ (\lambda i.\Gamma\ (\lambda j.f_{i,j})\ p_i)\ m)\ \ (\Gamma\ (\lambda i.\Gamma\ (\lambda j.g_{i,j})\ q_i)\ n)$$

With this denotation of the parts of *xss* and *yss*, we analyze and transform the body of `polymul`. The first step is to transform the outer nesting level of the result list into Γ-form. We use exp_k as an abbreviation for element k of this list. Remember that the list comprehension

$$[\ exp_k \mid k \leftarrow [0\,\texttt{..}\,m+n-2]\ ]$$

has been desugared by an earlier compiler phase into:

$$\texttt{map}\ (\lambda k.exp_k)\ [0\,\texttt{..}\,m+n-2]$$

Applying the rules of the Γ-calculus to the desugared form yields:

$$\Gamma\ (\lambda k.exp_k)\ (m+n-1)$$

Next, we look at the transformation of exp_k and infer the length of:

$$\texttt{sumN}\ [\ \texttt{mul}\ (xss\,\texttt{!!}\,i)\ (yss\,\texttt{!!}\,(k-i)) \mid i \leftarrow [low\,\texttt{..}\,high]\ ]$$

After desugaring and translation into Γ-form, we have:

$$\texttt{sumN}\ (\Gamma\ (\lambda i.\texttt{mul}\ (\Gamma\ (\lambda j.f_{i,j})\ p_i)\ \ (\Gamma\ (\lambda j.g_{k-i,j})\ q_{k-i}))\ (high-low+1))$$

We skip a lot of formal treatment here and present directly the size r of the result of the `sumN` application:

$$high-low+\texttt{Reduce}\ (\texttt{Maximum}, i, 0, high-low, \texttt{BTrue})(p_{(low+i)} + q_{(k-(low+i))})$$

6.4 Benefit of the Transformation

We have inferred that the result polynomial has $m+n-1$ coefficients, and coefficient k can be represented by r digits with r as stated above. If r is not simplified, its value must be computed quickly at run time. Function h is based on the recursive function `sumN` and cannot be stated as a simple closed expression since it depends on many run-time values. Inlining of `sumN` could, in principle, be done after this transformation, but very likely `sumN` will not be inlined due to its complexity.

The computation of the digits of each coefficient k remains the task of function `sumN`. However, the size r is sufficient for our purpose, since we can allocate the memory for the coefficients of the result in advance:

1. In a parallel computation of the coefficients, the number of bytes to be allocated for each communication buffer is known in advance. Thus, an appropriate representation assumed, the final location of a coefficient can already be used to receive the message.
2. A simulator of a digital design can statically allocate the exact amount of memory cells required to store the values, i.e., no dynamic data structures are required. A compiler which produces a hardware design has knowledge of the exact amount of bits required for each coefficient, if the values of the structural parameters are fixed.

7 Summary and Perspectives

Size inference enhances the possibilities for a compilation of functional programs of high efficiency. Obviously, if a list can be represented by an array because its length is known in advance, at least the amount of space for chaining the elements can be saved. Loss of dependences increases the potential for parallelization.

Often, intermediate copies of data objects can be saved since the result can be put immediately at the place where it is required. This makes communication more efficient.

The analysis is inherently undecidable and must be based on heuristics, e.g., partial evaluation, constraint solving, solving of recurrence equations, simplification of symbolic expressions and pattern matching with unification. We are pursuing the following strategy, which we hold to be quite promising: iterate alternatingly through applying size inference and then exploiting the results via program transformations. Possible transformations are inlining, fusion, deforestation and program specialization. The iterative process propagates static information successively deeper into the program structure, until the effort to evaluate the symbolic expressions at run time exceeds the gain.

We are going to implement size inference and the list transformation into our compiler. We have not been able to find a tool which provides adequate support for simplification of size expressions as we require; we may have to implement the simplifier ourselves. Furthermore, the compiler is undergoing a redesign in which the front end is being replaced by the front end of the Glasgow Haskell compiler.

Acknowledgements

We are grateful to Peter Faber, Paul Feautrier, John O'Donnell and Gregor Snelting for fruitful discussions. Michael Mendler, Walid Taha and the anonymous referees gave us valuable hints about related work. The work was supported by the German Research Foundation (DFG).

References

1. Jeffrey M. Bell, Françoise Bellegarde, and James Hook. Type-driven defunctionalization. *SIGPLAN Notices*, 32(8):25–37, 1997.
2. Gianni Bellè and Eugenio Moggi. Typed intermediate languages for shape-analysis. In *Typed Lambda Calculi and Applications (TLCA'97)*, Lecture Notes in Computer Science 1210, pages 11–29. Springer-Verlag, 1997.
3. Richard S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122–126, 1989.
4. Siddharta Chatterjee, Guy E. Blelloch, and Allan L.Fisher. Size and access inference for data-parallel programs. Technical Report CMU-CS-91-118, Dept. Computer Science, Carnegie-Mellon Univ., 1991.

5. Wei-Ngan Chin and Siau-Cheng Khoo. Tupling functions with multiple recursion parameters. In Patrick Cousot, Moreno Falaschi, Gilberto File, and Antoine Rauzy, editors, *Static Analysis Third Int. Workshop (WAS'93)*, Lecture Notes in Computer Science 724, pages 124–140. Springer-Verlag, 1993.
6. Wei-Ngan Chin and Siau-Cheng Khoo. Calculating sized types. In *Proceedings of the 2000 ACM SIGPLAN Workshop on Evaluation and Semantics-Based Program Manipulation (PEPM-00)*, pages 62–72, N.Y., 2000. ACM Press.
7. Koen Claessen and Mary Sheeran. A tutorial on Lava: A hardware description and verification system. Technical report, Dept. of Computing Science, Chalmers University of Technology, August 2000.
8. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Proc. 4th ACM Symp. Principles of Programming Languages (POPL'77)*, pages 238–252. ACM Press, 1977.
9. John Darlington, Anthony Field, Peter Harrison, Paul Kelly, David Sharp, Qian Wu, and Ronald L. While. Parallel programming using skeleton functions. In Arndt Bode, Mike Reeve, and Gottfried Wolf, editors, *PARLE'93: Parallel Architectures and Languages Europe*, Lecture Notes in Computer Science 694, pages 146–160. Springer-Verlag, 1993.
10. Nancy A. Day, Jeffrey R. Lewis, and Byrin Cook. Symbolic simulation of microprocessor models using type classes in Haskell. Technical Report CSE-99-005, Pacific Software Research Center, Oregon Graduate Institute, 1999.
11. Roberto Di Cosmo. *Isomorphisms of Types: from λ-calculus to information retrieval and language design*. Progress in Theoretical Computer Science. Birkhäuser, 1995.
12. Pascal Fradet and Julien Mallet. Compilation of a specialized functional language for massively parallel computers. Technical Report 3894, Institut National der Recherche en Informatique et en Automatique (INRIA), 2000.
13. Sergei Gorlatch. Extracting and implementing list homomorphisms in parallel program development. *Science of Computer Programming*, 33(1):1–27, 1999.
14. Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley Publishing Company, 1994.
15. Per Hammarlund and Björn Lisper. On the relation between functional and data parallel programming languages. In *Proc. Sixth Conf. on Functional Programming Languages and Computer Architecture (FPCA'93)*, pages 210–222. ACM Press, 1993.
16. haskell.org, 2000. `http://www.haskell.org`.
17. Christoph A. Herrmann and Christian Lengauer. Size inference of nested lists in functional programs. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proc. 10th Int. Workshop on the Implementation of Functional Languages (IFL'98)*, pages 346–364. Dept. Computer Science, Univ. College London, 1998.
18. Christoph A. Herrmann, Christian Lengauer, Robert Günz, Jan Laitenberger, and Christian Schaller. A compiler for $\mathcal{HDC}$. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Univ. Passau, May 1999.
19. Christoph A. Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, Fakultät für Mathematik und Informatik, Univ. Passau. Logos-Verlag, Berlin, 2000.
20. Christoph A. Herrmann and Christian Lengauer. $\mathcal{HDC}$: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.

21. Martin Hofmann. A type system for bounded space and functional in-place update – extended abstract. In Gerd Smolka, editor, *Programming Languages and Systems (ESOP 2000)*, Lecture Notes in Computer Science 1782, pages 165–179. Springer-Verlag, 2000.
22. John Hughes, Lars Pareto, and Amr Sabry. Proving the correctness of reactive systems using sized types. In *Proc. 23rd Ann. ACM Symp. on Principles of Programming Languages (POPL'96)*, pages 410–423. ACM Press, 1996.
23. C. Barry Jay and Milan Sekanina. Shape checking of array programs. In J. Harland, editor, *Proc. Australasian Theory Seminar on Computing*, volume 19 of *Australian Computer Science Communications*, pages 113–121, 1997.
24. Björn Lisper and Jean-François Collard. Extent analysis of data fields. In Baudouin Le Charlier, editor, *Static Analysis (SAS'94)*, Lecture Notes in Computer Science 864, pages 208–222. Springer-Verlag, 1994.
25. Hans-Wolfgang Loidl and Kevin Hammond. A sized time system for a parallel functional language. In *Proc. 1996 Glasgow Workshop on Functional Programming*, 1996. Electronic publication: `http://www.dcs.gla.ac.uk/fp/workshops/fpw96/Proceedings96.html`.
26. Jayadev Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Programming Languages and Systems*, 16(6):1737–1767, November 1994.
27. Flemming Nielson and Hanne Riis Nielson. *Two-level functional languages*. Number 34 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1992.
28. John T. O'Donnell. Bidirectional `fold` and `scan`. In *Functional Programming: Glasgow 1993*, Workshops in Computing, pages 193–200. Springer-Verlag, 1994.
29. John T. O'Donnell. From transistors to computer architecture: Teaching functional circuit specification in Hydra. In Pieter H. Hartel and Rinus Plasmeijer, editors, *Functional Programming Languages in Education*, Lecture Notes in Computer Science 1022, pages 195–214. Springer-Verlag, 1995.
30. James Riely and Jan Prins. Flattening is an improvement. In Jens Palsberg, editor, *Static Analysis (SAS'2000)*, Lecture Notes in Computer Science 1824, pages 360–376. Springer-Verlag, 2000.
31. Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proc. 14th ACM Symp. Principles of Programming Languages (POPL'87)*, pages 307–313. ACM Press, 1987.
32. Hongwei Xi and Frank Pfenning. Dependent types in practical programming. In *Proc. 26th ACM Symp. Principles of Programming Languages (POPL'99)*, pages 214–227. ACM Press, 1999.