An algebraic framework for parallelizing recurrence in functional programming

Rodrigo C. O. Rocha[12], Luís F. W. Góes[1], Fernando M. Q. Pereira[2]

[1]Institute of Exact Sciences and Informatics, PUC Minas
[2]Department of Computer Science, UFMG
{rcor,fernando}@dcc.ufmg.br, lfwgoes@pucminas.br

Abstract. The main challenge faced by automatic parallelization tools in functional languages is the fact that parallelism is often hidden under the syntax of complex recursive functions. In this paper, we propose an algebraic framework for parallelizing – automatically – two special classes of recursive functions. We show that these classes are comprehensive enough to include several well-known instances. We have used our ideas to implement a source-to-source compiler in Python to parallelize Haskell code. We have applied this prototype onto six different recursive functions, achieving, on a 4-core machine, speedups of up to 2.7x.

Keywords: recursive functions, parallel computing, functional programming, algebraic framework, abstract algebra

1 Introduction

The advent of multi-core computers has greatly spread the use of parallel programming among application developers. Yet, writing code that runs in parallel is still a difficult and error-prone task. Thus, the automatic parallelization of code has surfaced as an effective alternative to the development of high-performant programs [11, 18, 25]. In this sense, functional programming languages appear as a promising alternative to the development of parallel code. They provide referential transparency, reducing shared data and eliminating side effects, which makes automatic parallelization much easier. However, in spite of years of research, automatic generation of parallelism, out of functional code, is not a solved problem [12, 17]. Testimony of this last statement is the fact that functional code is still manually parallelized, usually by means of parallel skeletons [4, 6, 17].

The main challenge faced by automatic parallelization tools in functional languages is the fact that parallelism is often hidden under the syntax of complex recursive functions. There are several techniques to discover parallelism, such as work targeting list homomorphisms [5, 13, 20, 15], or the work of Fisher and Ghuloum [8], who parallelize imperative loops that can be translated as composition of functions. Nevertheless, the programming languages community still lacks approaches to infer parallelism on recursive functions automatically. The goal of this paper is to contribute to solve this omission by extending the family of recursive functions that can be parallelized automatically.

To achieve this objective, we propose an algebraic framework for parallelizing two special classes of recursive functions. These functions need to have two core properties. First: the recursive function must contain only operations that can be used to define monoids or semirings. Second: the propagation of arguments between recursive calls has to be defined by an invertible function. The proposed framework is based on the theory introduced by Fisher and Ghuloum [8]. Those authors have designed and tested an approach to parallelize imperative loops by transforming them in recurrence relations defined by the compositions of associative functions. We go beyond the work of Fisher and Ghuloum in two ways: (i) we work on recursive functions, instead of imperative loops; (ii) we provide a more general definition of parallel function composition. The key idea behind our findings is the fact that algebraic structures such as groups, monoids and semirings let us decompose recursive functions into simpler components, which are amenable to automatic parallelization.

To validate the ideas discussed in this paper, we have used them to implement a source-to-source compiler, in Python, that performs automatic parallelization of Haskell code. In Section 4 we show how to parallelize six different – and well-known – recursive functions. Key to efficiency is the fact that we can transform elements in the family of parallelizable functions into list homomorphisms. This transformation, which we explain in Section 3, to the best of our knowledge, is novel. Our experiments show that our technique is effective and useful. We have achieved speedups of up to 2.7x in a 4-core Intel processor. These results are even more meaningful if we consider that they have been obtained in a completely automatic way.

2 Overview

We will use the well-known factorial function as an example to introduce our ideas to the reader. This function can be defined as follows:

$$f(x) \coloneqq \begin{cases} 1 & \text{if } x = 1 \\ x \cdot f(x-1) & \text{otherwise} \end{cases}$$

Factorial is a very simple function, and the reader familiar with the parallelization of reductions on commutative and associative operators will know immediately that this function has a very efficient parallel implementation. Key to perform this parallelization is the observation that factorial can be re-written as a sequence of multiplications, e.g.: $f(x) = x \cdot (x-1) \cdot \ldots \cdot 2 \cdot 1$.

A key property of multiplication – associativity – lets us solve them in a pairwise fashion. This possibility gives us the chance to run the above expression in $O(\ln x)$ time. The goal of this paper is to be able to apply this kind of parallelization automatically onto recursive functions. In order to achieve this objective, we shall be re-writing functions as a composition of simpler functions which are associative. In the case of factorial, this composition looks like:

$$f(x) = (f'_{x-1} \circ \ldots \circ f'_2 \circ f'_1)(1)$$

How do we find a suitable implementation of f'_i? We first provide this answer for a family of recursive functions which have the following format:

$$f(p) \coloneqq \begin{cases} g_0(p) & \text{if } p = p_0 \\ g_1(p) \oplus f(h(p)) \oplus g_2(p) & \text{otherwise} \end{cases} \tag{1}$$

For any function f that can be written in the format above, we show that it is possible to decompose f into a composition of functions. We first identify h, the function used to propagate the arguments of f, which we call the *hop* function. The *hop* function must be a well-defined monotonic function, which must have an inverse. For the factorial example, we have the following *hop* function h: $h(x) = x - 1$, with inverse $h^{-1}(x) = x + 1$.

The *hop* function is fundamental for generating the next arguments of the sequence of compositions. Because it has an inverse, we use it to know $x-1$, the number of functions which will constitute the sequence of compositions. We find out $x - 1$ after solving the following equation: $p_0 = h^{x-1}(p)$. For the factorial example, the depth is exactly the initial argument x. After identifying the *hop* function and its inverse, we re-write f in a manner suitable for the composition of functions which does not contain a recursive call, e.g.: $f'_i(s) \coloneqq (i+1) \cdot s$, where s is the usually called accumulator parameter in funcional composition.

Now, we can write f such as $f(x) = (f'_{x-1} \circ f'_{x-2} \circ \cdots \circ f'_2 \circ f'_1)(1)$. This transformation is useful, considering that functional composition is an associative operation, which can often be parallelized if it is possible to symbolically compute and simplify intermediate compositions [8]. For instance, for $i > 1 \in \mathbb{Z}$, $(f'_i \circ f'_{i-1})(s) = (i+1) \cdot (i \cdot s)$ can be reassociated as $(f'_i \circ f'_{i-1})(s) = ((i+1) \cdot i) \cdot s$ which is essentially equivalent to the original function regarding computational complexity. Thus we can evaluate $f(x)$ by computing a reduction over a list of functions, using the functional composition operator, i.e., $f(x) = \circ/[f'_{x-1}, f'_{x-2}, \ldots, f'_1]$.

Section 3.2 generalizes the factorial example seen in this section. In Section 3.3 we extend our framework a bit further, showing that we can also parallelize functions in the format below. In this case, we consider two operators $\oplus$ and $\odot$, for which we only require that $\odot$ be associative, and $\oplus$ be commutative, in addition of being associative. Again, the hop function h must have an inverse function which can be computed efficiently.

$$f(p) \coloneqq \begin{cases} g_0(p) & \text{if } p = p_0 \\ g_1(p) \oplus (g_3(p) \odot f(h(p)) \odot g_4(p)) \oplus g_2(p) & \text{otherwise} \end{cases} \tag{2}$$

3 Automatic Parallelization of Recursive Functions

In this section we formalize the developments earlier seen in Section 2. To this end, we provide a few basic notions in Section 3.1. In Section 3.2, we show how to parallelize functions on the format given by Equation 1. In Section 3.3, we move on to deal with functions defined by Equation 2.

3.1 Technical Background

In the rest of this paper we shall use three notions borrowed from abstract algebra: *groups*, *monoids* and *semirings*. If S is a set, then we define these algebraic structures as follows:

- A group $G = (S, +)$ is a nonempty set closed under an associative binary-operation $+$, which is associative, invertible and has a zero element $\mathbf{0}$, the identity regarding $+$. For each $a \in S$, its inverse $-a$ also belongs to S. A group need not be commutative. If a group is commutative, it is usually called an abelian group [21].
- A monoid $M = (S, +)$ is a nonempty set closed under an associative binary-operation $+$ with identity $\mathbf{0}$. A monoid need not be commutative and its elements need not have inverses within the monoid.
- A semiring $R = (S, +, \cdot)$ is a nonempty set closed under two associative binary-operations $+$ and $\cdot$, called addition and multiplication, respectively [10, 9]. A semiring satisfies the following conditions:
  - $(S, +)$ is a commutative monoid with identity element $\mathbf{0}$;
  - $(S, \cdot)$ is a monoid with identity element $\mathbf{1}$;
  - Multiplication distributes over addition from either side;
  - Multiplication by $\mathbf{0}$ annihilates R, i.e. $a \cdot \mathbf{0} = \mathbf{0} \cdot a = \mathbf{0}$, for all $a \in S$.

Parallelizing functional composition. Functional composition is an associative binary operator over functions. Previous work [8, 19] has shown that, given a family of indexed functions $\mathcal{F}$ closed under functional composition $\circ$, a function $\psi : \mathbb{Z} \times \mathbb{Z} \to \mathcal{F}$ is a composition evaluator of $\mathcal{F}$ iff $i \psi j = f_i \circ f_j$, for $f_i, f_j \in \mathcal{F}$. If each function in $\mathcal{F}$ and the composition evaluator ψ are constant-time computations, then a sequence of n compositions can be efficiently computed in $O(n/p + \ln p)$, considering p processing units. A functional composition over $\mathcal{F}$ can be evaluated by using the composition evaluator ψ. Thus, given a sequence of compositions $f_n \circ f_{n-1} \circ \cdots \circ f_1$, this sequence can be efficiently computed using a reduction operator $\circ/[f_n, f_{n-1}, \ldots, f_1]$, since the following equivalence holds: $\circ/[f_n, f_{n-1}, \ldots, f_1] = \psi/[n, n-1, \ldots, 1]$.

3.2 Monoids

In this section, we generalize the solution presented in Section 2. We provide the formal description of the mechanism used for parallelizing the factorial recursive function, regarding general algebraic structures of groups and monoids. Let S_G and S_M be sets and $M = (S_M, +)$ a monoid. Let $f : S_G \to S_M$ be a recursive function defined as:

$$f(x) \coloneqq \begin{cases} g_0(x_0) & \text{if } x = x_0 \\ g_1(x) + f(h(x)) + g_2(x) & \text{otherwise} \end{cases}$$

We assume that each $g_i : S_G \to S_M$ are pure and non-recursive functions, i.e. if $a \in S_G$ then $g_i(a) \in S_M$, for $i \in [0, 2]$. Furthermore, we assume that the *hop* function $h : S_G \to S_G$ is an invertible and monotonic function over S_G.

Proposition 1 *The recursive function $f : S_G \to S_M$ can be written as a functional composition.*

Proof. We let $f'_i : S_M \to S_M$ be the following non-recursive function:

$$f'_i(s) = g_1((h^{-1})^i(x_0)) + s + g_2((h^{-1})^i(x_0))$$

In the above definition, we let $h^{-1} : S_G \to S_G$ be the inverse function of the hop h. Function $(h^{-1})^i(x_0)$ is the i-th functional power of $h^{-1} : S_G \to S_G$. To transform the recursive function into a composition, it is important to infer the depth of the recursive stack. Let $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$. Thus $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$ □

From Fisher and Ghuloum [8], we know that the composition of f'_i can be computed in parallel since, for $i > 1 \in \mathbb{Z}$, we have that:

$$\begin{aligned}
&(f'_i \circ f'_{i-1})(s) \Leftrightarrow g_1((h^{-1})^i(x_0)) + f'_{i-1}(s) + g_2((h^{-1})^i(x_0)) \Leftrightarrow \\
&g_1((h^{-1})^i(x_0)) + [g_1((h^{-1})^{i-1}(x_0)) + s + g_2((h^{-1})^{i-1}(x_0))] + g_2((h^{-1})^i(x_0)) \Leftrightarrow \\
&[g_1((h^{-1})^i(x_0)) + g_1((h^{-1})^{i-1}(x_0))] + s + [g_2((h^{-1})^{i-1}(x_0)) + g_2((h^{-1})^i(x_0))]
\end{aligned}$$

Defining the hop function. Let $G = (S_G, +)$ be a group with identity $\mathbf{0}$. If the *hop* function h is an invertible and monotonic function, we can calculate k. Let h be generally defined as $h(x) = e_1 + x + e_2$, where $e_1, e_2 \in S_G$. Then

$$\begin{aligned}
h^k(x) &= x_0 \Leftrightarrow \\
e_1 + \cdots + e_1 + e_1 + x + e_2 + e_2 + \cdots + e_2 &= x_0 \Leftrightarrow \\
ke_1 + x + ke_2 &= x_0 \Leftrightarrow \\
x + ke_2 &= (-ke_1) + x_0 \Leftrightarrow \\
x + ke_2 - x_0 &= (-ke_1) \Leftrightarrow \\
x + (ke_2 - x_0 + ke_1) &= \mathbf{0}
\end{aligned}$$

that is, $\exists k \in \mathbb{Z}$ such that $k > 0$ and $(ke_2 - x_0 + ke_1) \in S_G$ is the inverse of $x \in S_G$. Given the equality above, k can often be dynamically computed with a small overhead. If G is commutative, then $(ke_2 - x_0 + ke_1) = k(e_1 + e_2) - x_0$. From these notions, we define function h^{-1}, the inverse of the hopping function, as $h^{-1}(x) = (-e_1) + x + (-e_2)$. This equality is true, since:

$$\begin{aligned}
&(h^{-1} \circ h)(x) = (-e_1) + h(x) + (-e_2) \Leftrightarrow \\
&(h^{-1} \circ h)(x) = (-e_1) + [e_1 + x + e_2] + (-e_2) \Leftrightarrow \\
&(h^{-1} \circ h)(x) = \mathbf{0} + x + \mathbf{0} = x
\end{aligned}$$

Computing the function composition using list homomorphism. Thus far, we have seen how to re-write a function (with certain properties) as a composition of non-recursive functions. We need now a way to implement this composition efficiently. To achieve efficiency, we use list homomorphisms. We say that a function y is a list homomorphism if we have that $y(w +\!\!+\, z) = y(w) +\!\!+\, y(z)$, where $+\!\!+$ denotes list concatenation. In this section we derive a simple implementation for such a recursive function $f : S_G \to S_M$ by means of list homomorphism.

Proposition 2 *Let $h'(i) = (h^{-1})^i(x_0)$. Then we can write the functional composition $f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$ as:*

$$f(x) = (+/(g_1 \circ h') \star [k, k-1, \ldots, 1]) + g_0(x_0) + (+/(g_2 \circ h') \star [1, 2, \ldots, k]) .$$

Proof. The functional composition expands as follows:

$$\begin{aligned} f(x) &= g'_1(x_0) + g_0(x_0) + g'_2(x_0), \text{where:} \\ g'_1(x_0) &= g_1((h^{-1})^k(x_0)) + g_1((h^{-1})^{k-1}(x_0)) + \cdots + g_1((h^{-1})(x_0)) \\ g'_2(x_0) &= g_2((h^{-1})(x_0)) + g_2((h^{-1})^2(x_0)) + \cdots + g_2((h^{-1})^k(x_0)) \end{aligned}$$

Since $h'(i) = (h^{-1})^i(x_0)$, then we can write:

$$\begin{aligned} g'_1(x_0) &= g_1(h'(k)) + g_1(h'(k-1)) + \cdots + g_1(h'(1)) \\ g'_2(x_0) &= g_2(h'(1)) + g_2(h'(2)) + \cdots + g_2(h'(k)) \end{aligned}$$

Therefore, it is possible to compute g'_1 and g'_2 using a list homomorphism, i.e.:

$$\begin{aligned} g'_1(x_0) &= +/g_1 \star (h' \star [k, k-1, \ldots, 1]) \\ g'_2(x_0) &= +/g_2 \star (h' \star [1, 2, \ldots, k]) \end{aligned}$$

Where $+/\ell$ denotes the folding of the operation $+$ onto the list ℓ, and $k\star\ell$ denotes the mapping of function k onto every element of ℓ. The above expression can then be simplified as:

$$\begin{aligned} g'_1(x_0) &= +/(g_1 \circ h') \star [k, k-1, \ldots, 1] \\ g'_2(x_0) &= +/(g_2 \circ h') \star [1, 2, \ldots, k] \end{aligned}$$

□

3.3 Semirings

We now describe a second family of recursive functions that we can parallelize automatically by rethinking them under the light of algebraic structures. Let S_G and S_R be sets and $R = (S_R, +, \cdot)$ a semiring. Let $f : S_G \to S_R$ be defined as:

$$f(x) := \begin{cases} g_0(x_0) & \text{if } x = x_0 \\ g_1(x) + g_3(x) f(h(x)) g_4(x) + g_2(x) & \text{otherwise} \end{cases}$$

where each $g_i : S_G \to S_R$ is a non-recursive function. Function $h : S_G \to S_G$ is the *hop*. Let $f'_i : S_R \to S_R$ be the following non-recursive function:

$$f'_i(s) = g_1((h^{-1})^i(x_0)) + g_3((h^{-1})^i(x_0)) s g_4((h^{-1})^i(x_0)) + g_2((h^{-1})^i(x_0))$$

where $(h^{-1})^i(x_0)$ is the i-th functional power of h^{-1}, the inverse function of the *hop* function h (see Section 3.2).

In order to transform the recursive function into a composition, again we must infer the depth of the recursive stack. Let $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$ (see Section 3.2). Thus:

$$f(x) = (f'_k \circ f'_{k-1} \circ \cdots \circ f'_2 \circ f'_1)(g_0(x_0))$$

Computing the function composition using list homomorphism . In this section we derive a simple implementation of a recursive function $f : S_G \rightarrow S_R$ by means of list homomorphism. If $h'(i) = (h^{-1})^i(x_0)$, then we can write:

$$f(x) = \phi_1(k) + \phi_3(k) g_0(x_0) \phi_4(k) + \phi_2(k)$$

where $k > 0 \in \mathbb{Z}$ such that $h^k(x) = x_0$ (see Section 3.2). We have that:

$$\begin{aligned}
\beta_1(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1])\, g_1(h'(i))\, (\cdot/(g_4 \circ h') \star [i+1, i+2, \ldots, k]) \\
\phi_1(k) &= g_1(h'(k)) + (+/\beta_1 \star [k-1, k-2, \ldots, 1]) \\
\phi_3(k) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, 1]) \\
\phi_4(k) &= (\cdot/(g_4 \circ h') \star [1, 2, \ldots, k]) \\
\beta_2(i) &= (\cdot/(g_3 \circ h') \star [k, k-1, \ldots, i+1])\, g_2(h'(i))\, (\cdot/(g_4 \circ h') \star [i+1, i+2, \ldots, k]) \\
\phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, k-1]) + g_2(h'(k))
\end{aligned}$$

There are redundant computations in the previous definition. We can optimize the computation of β_1 and β_2 by pre-computing these redundant values using a scan operation. Considering left- and right-associative scan operations (*scanl* and *scanr*), we define lists v and w as follows:

$$\begin{aligned}
[v_1, v_2, \ldots, v_{k-1}] &= scanr \cdot /(g_3 \circ h') \star [k, k-1, \ldots, 2] \\
[w_1, w_2, \ldots, w_{k-1}] &= scanl \cdot /(g_4 \circ h') \star [2, 3, \ldots, k]
\end{aligned}$$

That is, $v_i = \cdot/(g_3 \circ h') \star [k, k-1, \ldots, k-i+1]$ and $w_i = \cdot/(g_4 \circ h') \star [k-i+1, \ldots, k-1, k]$. Once we have pre-computed v_i and w_i, we can define the following simplified construction:

$$\begin{aligned}
\beta_1(i) &= v_i \cdot g_1(h'(i)) \cdot w_{k-i} \\
\phi_1(k) &= g_1(h'(k)) + (+/\beta_1 \star [k-1, k-2, \ldots, 1]) \\
\phi_3(k) &= v_1 \cdot (g_3 \circ h')(1) \\
\phi_4(k) &= (g_4 \circ h')(1) \cdot w_{k-1} \\
\beta_2(i) &= v_i \cdot g_2(h'(i)) \cdot w_{k-i} \\
\phi_2(k) &= (+/\beta_2 \star [1, 2, \ldots, k-1]) + g_2(h'(k))
\end{aligned}$$

3.4 Examples

In this section we discuss different functions that we can parallelize automatically. We shall provide examples that we can parallelize using the monoid-based approach (Catalan Numbers and List Concatenation), and with the semiring-based approach (Financial Compound Interest, Horner's Method and Comb Filters). The actual performance of each of these example is analyzed in Section 4.

Catalan numbers. Catalan numbers form a sequence of positive integers that appear in the solution of several counting problems in combinatorics, including some generating functions. Catalan numbers are defined as follows:

$$C_n = \frac{2(2n-1)}{n+1} C_{n-1} \quad \text{where } C_1 = 1$$

which can be written as $f : \mathbb{Z} \to \mathbb{Z}^*$

$$f(x) \coloneqq \begin{cases} 1 & \text{if } x = 1 \\ \frac{2(2x-1)}{x+1} \cdot f(x-1) & \text{otherwise} \end{cases}$$

Similar to the factorial function seen in Section 2, the above function can be written as a composition of non-recursive functions: Let $h : \mathbb{Z} \to \mathbb{Z}$ be $h(x) = x - 1$, $g_1 : \mathbb{Z} \to \mathbb{Z}^*$ be $g_1(x) = \frac{2(2x-1)}{x+1}$ and $g_2 : \mathbb{Z} \to \mathbb{Z}^*$ be $g_2(x) = 1$.

Then, we can define a function $f'_i : \mathbb{Z}^* \to \mathbb{Z}^*$, such as

$$\begin{aligned} f'_i(s) &= g_1((h^{-1})^i(1)) \cdot s \cdot g_2((h^{-1})^i(1)) \Leftrightarrow \\ f'_i(s) &= g_1(i+1) \cdot s \cdot 1 \Leftrightarrow \\ f'_i(s) &= \frac{2(2i+1)}{i+2} \cdot s \end{aligned}$$

since $h^{-1}(x) = x+1$. We can easily calculate that $h^k(x) = 1$ for $k = x - 1$, since $x + (0k - 1 + k(-1)) = 0 \Leftrightarrow x - k - 1 = 0$.

Therefore, $f(x) = (f'_{x-1} \circ f'_{x-2} \circ \cdots \circ f'_2 \circ f'_1)(1)$. Since, for every $i > 1 \in \mathbb{Z}$, the composition $(f'_i \circ f'_{i-1})(s)$ can be symbolicaly computed and simplified, i.e.

$$\begin{aligned} (f'_i \circ f'_{i-1})(s) &= \frac{2(2i+1)}{i+2} \cdot f'_{i-1}(s) \Leftrightarrow \\ (f'_i \circ f'_{i-1})(s) &= \frac{2(2i+1)}{i+2} \cdot \left(\frac{2(2i-1)}{i+1} \cdot s \right) \Leftrightarrow \\ (f'_i \circ f'_{i-1})(s) &= \left(\frac{2(2i+1)}{i+2} \cdot \frac{2(2i-1)}{i+1} \right) \cdot s \end{aligned}$$

Function $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_1)(1)$ can be computed in parallel as a reduction.

List Concatenation. Let $L_\mathbb{Z}$ be the set of lists over the set of integers $\mathbb{Z}$. A list is an ordered sequence denoted by $A = [a_1, a_2, \ldots, a_n]$, where the size of A is $\#A = n$. An empty list is denoted by $[]$ and $\#[] = 0$. Concatenation is an associative binary-operation over a set of lists. Given two lists $A = [a_1, a_2, \ldots, a_n]$ and $B = [b_1, b_2, \ldots, b_m]$, the concatenation of the lists A and B is denoted by $A +\!\!+\ B = [a_1, \ldots, a_n, b_1, \ldots, b_m]$. The identity element regarding concatenation is the empty list. Thus$(L_\mathbb{Z}, +\!\!+\ )$ is a non-commutative monoid. Let $f : \mathbb{Z} \to L_\mathbb{Z}$ be the following recursive function:

$$f(x) \coloneqq \begin{cases} [] & \text{if } x = 0 \\ [x] +\!\!+\ f(x-1) +\!\!+\ [x] & \text{otherwise} \end{cases}$$

Let $h : \mathbb{Z} \to \mathbb{Z}$ be $h(x) = x - 1$, $g_1 : \mathbb{Z} \to L_\mathbb{Z}$ be $g_1(x) = [x]$ and $g_2 : \mathbb{Z} \to L_\mathbb{Z}$ be $g_2(x) = [x]$. Then, we can define a simplified function $f'_i : L_\mathbb{Z} \to L_\mathbb{Z}$, such as:

$$\begin{aligned} f'_i(s) &= g_1((h^{-1})^i(0)) +\!\!+\ s +\!\!+\ g_2((h^{-1})^i(0)) \Leftrightarrow \\ f'_i(s) &= g_1(i) +\!\!+\ s +\!\!+\ g_2(i) \Leftrightarrow \\ f'_i(s) &= [i] +\!\!+\ s +\!\!+\ [i] \end{aligned}$$

since $h^{-1}(x) = x + 1$. We have that $h^k(x) = 0$ for $k = x$, since $x + (0k - 0 + k(-1)) = 0 \Leftrightarrow x - k = 0$. Therefore, $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_2 \circ f'_1)([])$. Since, for every $i > 1 \in \mathbb{Z}$, the composition $(f'_i \circ f'_{i-1})(s)$ can be symbolicaly computed and simplified, i.e.:

$$\begin{aligned}
&(f'_i \circ f'_{i-1})(s) = [i] +\!\!+\ f'_{i-1}(s) +\!\!+\ [i] \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = [i] +\!\!+\ ([i-1] +\!\!+\ s +\!\!+\ [i-1]) +\!\!+\ [i] \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = ([i] +\!\!+\ [i-1]) +\!\!+\ s +\!\!+\ ([i-1] +\!\!+\ [i]) \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = [i, i-1] +\!\!+\ s +\!\!+\ [i-1, i]
\end{aligned}$$

Thus $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_1)([])$ can be computed in parallel by a reduction.

Financial Compound Interest. We can define financial compound interest with periodic deposits recursively. Let $f : \mathbb{Z} \to \mathbb{R}$ be the following recursive function:

$$f(x) \coloneqq \begin{cases} y_0 & \text{if } x = 0 \\ (1+r) \cdot f(x-1) + y_x & \text{otherwise} \end{cases}$$

where y_0 is the initial deposit, r is the compounded rate, and y_x is the deposit on the x-th period. Let $h : \mathbb{Z} \to \mathbb{Z}$ is $h(x) = x - 1$, $g_1 : \mathbb{Z} \to \mathbb{R}$ is $g_1(x) = 0$, $g_3 : \mathbb{Z} \to \mathbb{R}$ is $g_3(x) = (1+r)$, $g_4 : \mathbb{Z} \to \mathbb{R}$ is $g_4(x) = 1$, $g_2 : \mathbb{Z} \to \mathbb{R}$ is $g_2(x) = y_x$. From these notions, we define a simplified function $f'_i : \mathbb{R} \to \mathbb{R}$ as follows:

$$\begin{aligned}
&f'_i(s) = g_1((h^{-1})^i(0)) + g_3((h^{-1})^i(0)) s g_4((h^{-1})^i(0)) + g_2((h^{-1})^i(0)) \Leftrightarrow \\
&f'_i(s) = g_1(i) + g_3(i) s g_4(i) + g_2(i) \Leftrightarrow \\
&f'_i(s) = (1+r)s + y_i \Leftrightarrow
\end{aligned}$$

since $h^{-1}(x) = x + 1$. We have that $h^k(x) = 0$ for $k = x$, since $x + (0k - 0 + k(-1)) = 0 \Leftrightarrow x - k = 0$. Hence, $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_2 \circ f'_1)(y_0)$. Since, for every $i > 1 \in \mathbb{Z}$, the composition $(f'_i \circ f'_{i-1})(s)$ can be symbolicaly computed and simplified, i.e.:

$$\begin{aligned}
&(f'_i \circ f'_{i-1})(s) = (1+r) f'_{i-1}(s) + y_i \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = (1+r)((1+r)s + y_{i-1}) + y_i \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = (1+r)(1+r)s + (1+r)y_{i-1} + y_i \Leftrightarrow \\
&(f'_i \circ f'_{i-1})(s) = (1+r)^2 s + [(1+r)y_{i-1} + y_i]
\end{aligned}$$

Thus $f(x) = (f'_x \circ f'_{x-1} \circ \cdots \circ f'_2 \circ f'_1)(y_0)$ can be computed in parallel.

Horner's method. Horner's method is useful to solve polynomials defined recursively. Its implementation can be parallelized as done in the previous example of financial compound interest. Let c_i be the coefficients, for $0 \leq i \leq n$. Thus a polynomial of degree n can be evaluated, for a given value of x, by the following recursive formula, as described by the Horner's method:

$$f(n) \coloneqq \begin{cases} c_0 & \text{if } n = 0 \\ f(n-1) \cdot x + c_n & \text{otherwise} \end{cases}$$

Comb filter in signal processing. Comb filters have several applications in signal processing [23]. The following equation represents the feedback form used by comb filters: $y_t = \alpha y_{t-T} + (1-\alpha)x_t$, where x_t is the input signal at a given time t and α controls the intensity that the delayed signal is fed back into the output y_t given a delay time T. Let $f : \mathbb{Z} \to \mathbb{R}$ be the following recursive function:

$$f(t) \coloneqq \begin{cases} y_0 & \text{if } t = 0 \\ \alpha f(t-T) + (1-\alpha)x_t & \text{otherwise} \end{cases}$$

3.5 Implementation

We have used the ideas discussed in this paper to implement a prototype of our automatic parallelizer that performs source-to-source transformations. This prototype is written in Python, and it performs symbolic computations using Python's `sympy` package. Symbolic computing allows us to find the inverse of the hop function and also to infer the depth of the recursive stack (as discussed in Sections 3.2 and 3.3). Below we show the source-to-source transformation that we produce for the list concatenation example:

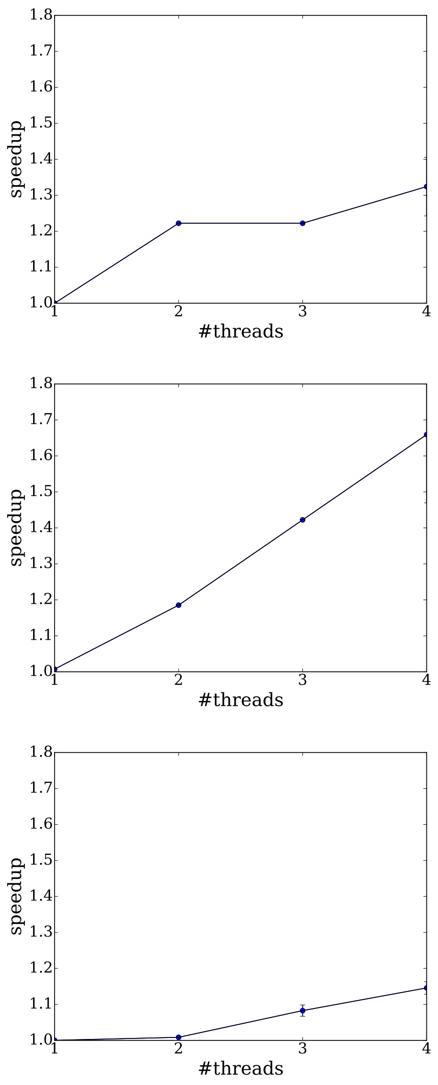
```
# Sequential version:
f :: Integer -> [Integer]
f 0 = []
f n = [n] ++ f(n-1) ++ [n]

# Parallel version:
f_g_1 :: Integer -> [Integer]
f_g_1 _HOP_i = [(_HOP_i)]
f_g_2 :: Integer -> [Integer]
f_g_2 _HOP_i = [(_HOP_i)]
f :: Integer -> [Integer]
f n = (parFoldr (++) (map f_g_1 (reverse [1..(n)]))) ++ [] ++
      (parFoldr (++) (map f_g_2 [1..(n)]))
```

In general, for the monoid-based case, we receive inputs in the format below:

```
f :: Integer -> IMGSET
f e_0 = y_0
f n = g_1(n) * f(n-e_1) * g_2(n)
```

The prototype's output consists of three new functions: `f_g_1`, `f_g_2` and `f`, which have the following general format:

```
f_g_1 :: Integer -> IMGSET
f_g_1 _HOP_i = g_1((_HOP_i*e_1 + e_0))
f_g_2 :: Integer -> IMGSET
f_g_2 _HOP_i = g_2((_HOP_i*e_1 + e_0))
f :: Integer -> IMGSET
f n = (parFoldr (*) (map f_g_1 (reverse [1..((-e_0 + n)/e_1)]))) * y_0 *
      (parFoldr (*) (map f_g_2 [1..((-e_0 + n)/e_1)]))
```



Factorial.

$$f(x) \coloneqq \begin{cases} 1 & \text{if } x = 1 \\ x \cdot f(x-1) & \text{otherwise} \end{cases}$$

Catalan.

$$f(x) \coloneqq \begin{cases} 1 & \text{if } x = 1 \\ \frac{2(2x-1)}{x+1} \cdot f(x-1) & \text{otherwise} \end{cases}$$

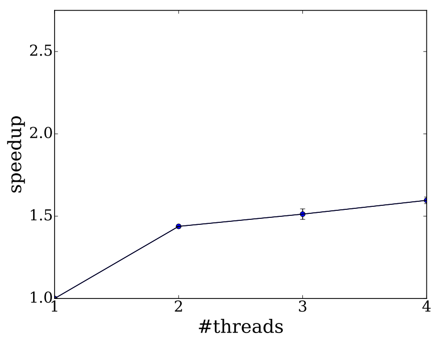
List concatenation.

$$f(x) \coloneqq \begin{cases} [] & \text{if } x = 0 \\ [x] +\!\!+\, f(x-1) +\!\!+\, [x] & \text{otherwise} \end{cases}$$

Fig. 1. Speedup analysis of monoid-based examples.

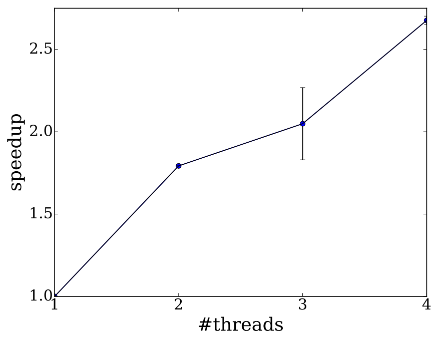
4 Evaluation

To validate the ideas discussed in this paper, we have implemented our source-to-source compiler that generates parallel Haskell, using the parallel library provided by the `Strategies` package, available in the Glasgow Haskell Compiler (GHC) [1, 16]. The experiments were performed in four physical cores of an Ivy Bridge-based Intel processor technology, with 2 GHz of clock and 15 GB of RAM. In this section, we show results for six benchmarks – three illustrating



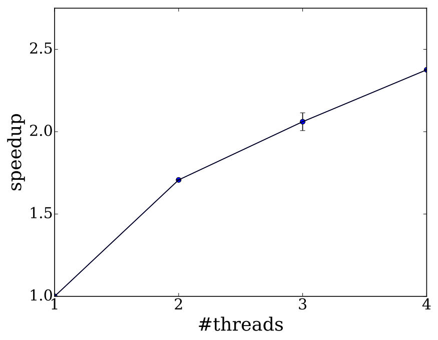
Financial compound interest.

$$f(x) \coloneqq \begin{cases} y_0 & \text{if } x = 0 \\ (1+r) \cdot f(x-1) + y_x & \text{otherwise} \end{cases}$$



Horner's method.

$$f(n) \coloneqq \begin{cases} c_0 & \text{if } n = 0 \\ f(n-1) \cdot x + c_n & \text{otherwise} \end{cases}$$



Comb filter.

$$f(t) \coloneqq \begin{cases} y_0 & \text{if } t = 0 \\ \alpha \cdot f(t-T) + (1-\alpha) \cdot x_t & \text{otherwise} \end{cases}$$

Fig. 2. Speedup analysis of semiring-based examples.

monoid-based parallelization (factorial, catalan and concatenation), and three illustrating semiring-based parallelization (compound interest, Horner's method and Comb Filter). For the experiments with all six applications we evaluated the speedup varying the number of threads from 1 to 4, while fixing the input argument of each function in 50,000. We consider a 95% confidence interval for a total of five executions.

Monoids. Figure 1 presents the results for the parallelization of recursive functions based on monoids. Vertical bars show confidence interval. For the parallelization of the three monoid-based applications we implemented the construction using the list homomorphism presented in Section 3.2. Parallelization was achieved by means of a right-associative fold operator, which is part of the Parallel Haskell library. We achieved a maximum speedup of 1.78× in the implementation of Catalan Numbers.

Semirings. Figure 2 shows the results for the parallelization of recursive functions based on semirings. Our largest speedup was 2.71 in the Horner's Method example. For the other two examples, we got more modest speedups. For the parallelization of the three semiring-based applications we implemented the construction using the list homomorphism discussed in Section 3.3. We have used the same parallel implementation of the right-associative fold operator which we applied on the monoid-based examples.

Discussion. We have been able to observe actual speedups on the six examples that we have played with. These speedups were usually sublinear, e.g., we could not observe a four-fold speedup in any of the cases. We believe that this sublinearity is due to the overhead imposed by the reduction operator required by the proposed parallel construction. Nevertheless, we would like to emphasize that all these results have been obtained by means of automatic transformations. In other words, the use of our techniques does not require any intervention from the programmer who has implemented the original version of each function that we parallelize.

5 Related Work

There are several automatic parallelization techniques that, similarly to ours, seek common patterns in code. Strategies based on matrix-multiplication are a well-known example. Kogge and Stone [14] have shown how to parallelize a recurrence equation by rewriting it in a form of matrix multiplication, also called *state-vector update* form. An expression e in a loop is a recurring expression if, and only if, e is computed from some loop-carried value. Sato and Iwasaki [22] have described a framework based on matrix multiplication for automatically parallelizing affine loops that consist of reduce or scan operations. They have also provided algorithms for recognizing the normal form and max-operators automatically. They have been able to report considerable speedups and high scalability by applying their framework onto simple benchmarks. Also along this line, Zou and Rajopadhye [26] have proposed a way to parallelize scan operations using the matrix multiplication framework with the polyhedral model [2, 3, 24]. They can handle arbitrary nested affine loops; the polyhedron model itself has already been used to parallelize different types of loops in imperative programming languages [7]. Contrary to our work, these previous approaches search for a way to deconstruct a loop as multiplication of matrices, we search for a way to deconstruct a function as a composition of monoid/semiring operations. The programs that can be parallelized by these two approaches are different.

Fisher and Ghuloum [8] provide a generalized formalization for automatic parallelization of loops by extracting function composition as the main associative operator. If a function is closed under composition, its compositions can be computed efficiently. They describe loops that compute reduction or scan as the composition of its *modeling function*. For loops that fit the allowed format, they can be implemented in a manner that computes the composition of the modelling function in parallel. Our work improves on theirs, because we extend their approach to recursive functions. In fact, this is our main contribution: a general way to extract parallelism buried under the syntax of potentially convoluted recursive functions. In addition, we also provide a more general definition of parallel code by means of algebraic structures such as monoids and semirings.

There exists vast literature about list homomorphisms [5, 13, 20, 15]. List homomorphism is a special class of natural recursive functions on lists, which has algebraic properties suitable for parallelism. We rely on list homomorphisms to build efficient parallel computations of recursive functions. However, our approach does not target exclusively functions that work on lists. As many of our examples illustrate, we are able to generate parallel code for functions involving just numbers, or even for more complex data-structures that can be processed by monoid-based operators.

6 Conclusion

This paper has presented a theoretical approach to parallelize recursive functions. This contribution is important because previous work has reported difficulties to infer parallel behavior out of recursive function. In this case, parallelism is usually buried under heavy and convoluted syntax. We have delineated two classes of recursive functions which we can parallelize automatically. These functions have the following property: they can be re-written as the combination of themselves (through a recursive call) with non-recursive functions by means of monoid or semiring operators. There are several examples of functions that fit this framework, including typical functional implementations of algorithms to compute factorials, sum up elements of lists, concatenate lists, etc.

As future work, we intend to broaden the classes of recursive functions that our algebraic framework can automatically parallelize. We also intend to perform optimizations on the parallel code generated by our source-to-source compiler.

References

1. Berthold, J., Marlow, S., Hammond, K., Al Zain, A.: Comparing and optimising parallel Haskell implementations for multicore machines. In: ADPNA. IEEE (2009)
2. Bondhugula, U., Baskaran, M., Krishnamoorthy, S., Ramanujam, J., Rountev, A., Sadayappan, P.: Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In: CC. Springer (2008)
3. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelizer and locality optimizer. In: PLDI. ACM (2008)

4. Brown, C., Danelutto, M., Hammond, K., Kilpatrick, P., Elliott, A.: Cost-directed refactoring for parallel erlang programs. International Journal of Parallel Programming 42(4) (2013)
5. Cole, M.: Parallel programming with list homomorphisms. Parallel Processing Letters 5(02) (1995)
6. Collins, A., Grewe, D., Grover, V., Lee, S., Susnea, A.: NOVA: A functional language for data parallelism. In: ARRAY. ACM (2014)
7. Feautrier, P.: Automatic parallelization in the polytope model. In: The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications. pp. 79–103. Springer (1996)
8. Fisher, A.L., Ghuloum, A.M.: Parallelizing complex scans and reductions. In: PLDI. ACM (1994)
9. Golan, J.S.: Power Algebras over Semirings: With Applications in Mathematics and Computer Science. Mathematics and Its Applications 488, Springer, 1 edn. (1999)
10. Golan, J.S.: Semirings and their Applications. Springer, 1 edn. (1999)
11. Govindarajan, R., Anantpur, J.: Runtime dependence computation and execution of loops on heterogeneous systems. In: CGO. IEEE/ACM (2013)
12. Hammond, K., Berthold, J., Loogen, R.: Automatic skeletons in template haskell. Parallel Processing Letters 13(03) (2003)
13. Hu, Z., Iwasaki, H., Takechi, M.: Formal derivation of efficient parallel programs by construction of list homomorphisms. Trans. Program. Lang. Syst. 19(3) (1997)
14. Kogge, P.M., Stone, H.S.: A parallel algorithm for the efficient solution of a general class of recurrence equations. Trans. Comput. 22(8) (1973)
15. Liu, Y., Hu, Z., Matsuzaki, K.: Towards systematic parallel programming over mapreduce. In: EuroPar. Lecture Notes in Computer Science, Springer (2011)
16. Marlow, S., Maier, P., Loidl, H.W., Aswad, M.K., Trinder, P.W.: Seq no more: Better strategies for parallel Haskell. In: Haskell Symposium. ACM Press (2010)
17. Marlow, S., Peyton Jones, S., Singh, S.: Runtime support for multicore haskell. In: ICFP. pp. 65–78. ACM (2009)
18. Misailovic, S., Kim, D., Rinard, M.: Parallelizing sequential programs with statistical accuracy tests. Trans. Embed. Comput. Syst. 12(2) (2013)
19. Morihata, A., Matsuzaki, K.: Automatic parallelization of recursive functions using quantifier elimination. In: FLOPS. Lecture Notes in Computer Science, Springer (2010)
20. Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs. In: PLDI. ACM (2007)
21. Rotman, J.J.: Advanced Modern Algebra. Prentice Hall, 2 edn. (2003)
22. Sato, S., Iwasaki, H.: Automatic parallelization via matrix multiplication. In: PLDI. ACM (2011)
23. Schlecht, S.J., Habets, E.A.P.: Connections between parallel and serial combinations of comb filters and feedback delay networks. In: IWAENC (2012)
24. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: PACT. IEEE (2009)
25. Wang, Z., Tournavitis, G., Franke, B., O'Boyle, M.F.P.: Integrating profile-driven parallelism detection and machine-learning-based mapping. Trans. Archit. Code Optim. 11(1) (2014)
26. Zou, Y., Rajopadhye, S.: Scan detection and parallelization in "inherently sequential" nested loop programs. In: CGO. ACM (2012)