A T2 Graph-Reduction Approach To Fusion

Troels Henriksen, Cosmin E. Oancea
HIPERFIT, Department of Computer Science, University of Copenhagen (DIKU)
athas@sigkill.dk, cosmin.oancea@diku.dk

Abstract

Fusion is one of the most important code transformations as it has the potential to substantially optimize both the memory hierarchy time overhead and, sometimes asymptotically, the space requirement. In functional languages, fusion is naturally and relatively easily derived as a producer-consumer relation between program constructs that expose a richer, higher-order algebra of program invariants, such as the `map-reduce` list homomorphisms.

In imperative languages, fusing producer-consumer loops requires dependency analysis on arrays applied at loop-nest level. Such analysis, however, has often been labeled as "heroic effort" and, if at all, is supported only in its simplest and most conservative form in industrial compilers.

Related implementations in the functional context typically apply fusion only when the to-be-fused producer is used exactly once, i.e., in the consumer. This guarantees that the transformation is conservative: the resulting program does not duplicate computation.

We show that the above restriction is more conservative than needed, and present a structural-analysis technique, inspired from the $\mathtt{T}_1$-$\mathtt{T}_2$ transformation for reducible data flow, that enables fusion even in some cases when the producer is used in different consumers *and* without duplicating computation.

We report an implementation of the fusion algorithm for a *functional*-core language, named $\mathcal{L}_0$, which is intended to support *nested* parallelism across *regular* multi-dimensional arrays. We succinctly describe $\mathcal{L}_0$'s semantics and the compiler infrastructure on which the fusion transformation relies, and present compiler-generated statistics related to fusion on a set of six benchmarks.

Categories and Subject Descriptors D.1.3 [*Concurrent Programming*]: Parallel Programming; D.3.4 [*Processors*]: Compiler

General Terms Performance, Design, Algorithms

Keywords fusion, autoparallelization, functional language

1. Introduction

One of the main goals of the HIPERFIT project has been to develop the infrastructure necessary to write real-world, big-data financial applications in a hardware-independent language that can be efficiently executed on massively parallel hardware, e.g., GPGPU.

In this sense we have examined several such computational kernels [26], originally implemented in languages such as OCaml, Python, C++, C and measuring in the range of hundreds of lines of compact code, with two main objectives in mind:

1. What should be a suitable core language that, on the one hand, would allow a relatively straight-forward code translation, and, on the other hand, would preserve the algorithmic invariants that are needed to optimize the application globally?
2. What compiler optimizations would result in efficiency comparable to code hand-tuned for the specific hardware?

The answer to the first question has been a *functional* language, dubbed $\mathcal{L}_0$, supporting `map-reduce` *nested* parallelism on *regular* arrays, i.e., each row of the array has the same size:
It is *regular* because our suite does not require irregular arrays in the sense of NESL or DPH [9, 13], and regular arrays are more amenable to compiler optimizations, e.g., they allow transposition and simplified size analysis. It is *nested* because our suite exhibits several layers of parallelism that cannot be exploited by flat parallelism in the style of REPA [22], e.g., several innermost `scan` or `reduce` operations and at least one (semantically) sequential loop per benchmark. Finally, it is *functional* because we would rather invest compiler effort in exploiting high-level program invariants rather than in proving them. The common example here is parallelism: `map-reduce` constructs are inherently parallel, while Fortran-style `do` loops require sophisticated analysis to decide parallelism. Furthermore, such analyses [12, 19, 27, 28] have not yet been integrated in the repertoire of commercial compilers, likely due to "heroic effort" concerns, albeit (i) their effectiveness was demonstrated on comprehensive suites, and (ii) some of them were developed more than a decade ago.

Perhaps less expectedly, the answer to the second question seems to be that a common ground needs to be found between functional and imperative optimizations and, to a lesser extent, between language constructs. Much in the same way in which (data) parallelism seems to be generated by a combination of `map`, `reduce`, and `scan` operations, the optimization opportunities, e.g., enhancing the degree of parallelism and reducing the memory time and space overheads, seem solvable via a combination of `fusion`, `transposition`, loop `interchange` and `distribution` [2]. It follows that loops are necessary in the intermediate representation, regardless of whether they are provided as a language construct or are derived from tail-recursive functions via a code transformation.

Finally, an indirect consequence of having to deal with sequential (dependent) loops is that $\mathcal{L}_0$ provides support for "in-place updates" of array elements. The semantics is the functional one, i.e., deep copy of the original array but with the corresponding element replaced, `intersected` with the imperative one, i.e., if aliasing may prevent an in-place implementation a compile-time error is signaled. This approach enables the intuitive (optimized) cost model that the user likely assumes, while preserving the functional semantics. Section 2 provides an overview of the $\mathcal{L}_0$ language and of the enabling optimizations that set the stage for fusion.

FHPC'13,, September 23, 2013, Boston, Massachusets..

ACM 978-1-4503-2381-9/13/09...$15.00.
http://dx.doi.org/10.1145/2502323.2502328

1.1 Fusion and Principal Contributions

The simplest form of fusion is the one that relies on the invariant that mapping each element of a list by a function `f`, and then mapping each element of the result with a function `g` is the same as mapping each element of the original list with the composition of `g` with `f`. In Haskell-like notation this can be written as: `map g . map f` $\equiv$ `map (g . f)`, where the semantics of `map` is `map f [`a_1`,..,`a_n`] = [f `a_1`,..,f `a_n`]`. One can already observe that fusion may significantly optimise computation since it may replace some of the array accesses with scalar accesses, e.g., if the element type is a basic type.

Furthermore, defining `reduce` as: `reduce` $\oplus$ `e [`a_1`,..,`a_n`]` $\equiv$ `e`$\oplus$`a`$_1\oplus\ldots\oplus$`a`$_n$, it is possible to fuse a `reduce` with a `map` as `reduce` $\oplus$ `e . map f` $\equiv$ `reduce` $\oplus$ `e . map (reduce` $\oplus$ `e . map f) . split`$_P$, i.e., the list is distributed between P processors, each processor executes sequentially its chunk, and at the end the result is reduced (in parallel) across processors [7]. One may observe that this fusion may asymptotically improve the space requirement, e.g., if the input array is the iteration space. The operator $\oplus$ must be associative with identity e. Another observation is that both `map` and `reduce` have well-known efficient parallel implementations and that fusion preserves parallelism.

While in a functional language the (richer) semantics of operators such as `map`, `reduce` makes fusion easy to understand and to implement by the compiler, this is not the case in an imperative context. For example, fusing two parallel loops requires to prove that the set of array indices written by iteration i of the producer loop, denoted W_i, is a superset of the set of indices read by the consumer loop, denoted R_i, and also that $W_i \cap R_j = \emptyset, \forall i \neq j$. This analysis quickly requires "heroic effort" when the loops' body exhibits non-affine indexing or non-trivial control flow, e.g., loop nests.

Current fusion work in the functional context takes two main directions: One approach is to perform fusion aggressively, i.e., even when it duplicates computation, and to provide the user with primitives that inhibit fusion [15, 22]. The other approach performs fusion conservatively by means of rewrite rules or skeletons [6, 13, 21]. Since the latter relies tightly on the inlining engine, e.g., to create exploitable patterns, these approaches typically do not fuse an array that is used multiple times.

Main contributions of this paper are:

- A structural analysis technique that succeeds to conservatively fuse, i.e., without duplicating computation, even when an array has multiple uses, *if* the dependency graph of the producer-consumer array combinators is reducible (via T_2 reduction). The compositional algebra for fusion includes the `map`, `reduce` and `filter` operators. These are presented in Section 3.
- An analysis refinement that enables fusion across (otherwise) "inhibitor" built-in functions such as `size`, `split`, `transpose`.
- Two other transformations (not implemented yet) that enable and are enabled by fusion and that optimize the use of `scan` and `reduce`. For example, ISWIM interchanges the outer-`scan` with an inner `map` nest, and thus enhances the (exploitable) parallelism degree of the program. These are discussed in Section 4.

Finally, Section 5 compares with related work and Section 6 concludes the paper.

2. Preliminaries: $\mathcal{L}_0$ and Enabling Optimizations

$\mathcal{L}_0$ is a mostly-monomorphic, statically typed, strictly evaluated, purely functional language. Although some built-in higher-order functions provide polymorphism, user-defined functions are monomorphic, and their return- and parameter types must be specified explicitly. For brevity, we will not cover the typing rules in detail, as they are quite trivial. The only exception is our notion of *uniqueness types*, which is given a cursory treatment in Section 2.2.

t	::=	`int`	(Integers)
	|	`real`	(Floats)
	|	`bool`	(Booleans)
	|	`char`	(Characters)
	|	$(t_1,\ \ldots,\ t_n)$	(Tuples)
	|	$[t]$	(Arrays)
v	::=	k	(Integer)
	|	x	(Decimal number)
	|	b	(Boolean)
	|	c	(Character)
	|	$(v_1, \ldots, v_n)$	(Tuple)
	|	$\{v_1, \ldots, v_n\}$	(Array)
p	::=	**id**	(Name pattern)
	|	$(p_1,\ \ldots,\ p_n)$	(Tuple pattern)
e	::=	v	(Constant)
	|	**id**	(Variable)
	|	$(e_1,\ldots,e_n)$	(Tuple expression)
	|	$\{e_1,\ldots,e_n\}$	(Array expression)
	|	$e_1 \odot e_2$	(Binary operator)
	|	$\sim e$	(Prefix minus)
	|	`not` e	(Logical negation)
	|	`if` e_1 `then` e_2 `else` e_3	(Branching)
	|	**id**$[e_1,\ \ldots,\ e_n]$	(Indexing)
	|	**id**$(e_1,\ \ldots,\ e_n)$	(Function call)
	|	`let` p `=` e_1 `in` e_2	(Pattern binding)
	|	`zip(`e_1`,` $\ldots$`,` e_n`)`	(Zipping)
	|	`unzip(`e`)`	(Unzipping)
	|	`iota(`e`)`	(Range)
	|	`replicate(`e_n`,` e_v`)`	(Replication)
	|	`size(`e`)`	(Array length)
	|	`transpose(`e`)`	(Transposition)
	|	`split(`e_1`,` e_2`)`	(Split e_2 at index e_1)
	|	`concat(`e_1`,` e_2`)`	(Concatenation)
	|	`let` α `=` β `with` $[e_1,\ldots,e_n]$ `<-` e_v `in` e_b	(In-place update)
	|	`loop (`p `=` e_1`) =` `for` α `<` e_2 `do` e_3 `in` e_4	(Loop)
fun	::=	`fun` t **id**$(t_1$ **id**$_1,\ldots t_n$ **id**$_n)$ `=` e	
$prog$	::=	ϵ	
	|	$fun\ prog$	

Figure 1. $\mathcal{L}_0$ syntax

The syntax of the first-order fragment of $\mathcal{L}_0$ can be seen in Figure 1. Common arithmetic operators such as `+` and `/` are supported in the usual way. Note that they are polymorphic in the sense that they accept both integers and floating-points, although both operands must be of the same type. Pattern matching is supported in a limited way as the only way of decomposing tuple values, but there is no `case` construct. Note that braces denote arrays, not sets.

`zip` and `unzip` behave as usual, i.e, `zip({1,2,3},{4,5,6})` $=$ `{(1,4),(2,5),(3,6)}`, but the semantics of `zip` requires that the input arrays have outermost dimensions of equal sizes. Otherwise a compile or runtime error is signalled.

There are two non-standard constructs in the first-order part of $\mathcal{L}_0$. The first is the let-with construct for updating parts of arrays:

```
let b = a with [i1,..,ik] <- v in body
```

```
loop (x = a) =                    fun t f(int i, int n, t x) =
  for i < n do                      if i >= n then x
    g(x)                       ⇒        else f(i+1, n, g(x))
in body                           let x = f(i, n, a)
                                  in body
```

Figure 2. Loop to recursive function

l	::=	`fn` t `(`t_1 **id**$_1$`, ..., `t_n **id**$_n$`)`	(Anonymous function)
		`=>` e	
	|	**id** `(`e_1`, ..., `e_n`)`	(Curried function)
	|	`op` $\odot$ `(`e_1`, ..., `e_n`)`	(Curried operator)
e	::=	`map(`l`, `e`)`	
	|	`filter(`l`, `e`)`	
	|	`reduce(`l`, `x`, `e`)`	
	|	`scan(`l`, `x`, `e`)`	
	|	`redomap(`l_r`, `l_m`, `x`, `e`)`	

Figure 3. Second-order array combinators

The above evaluates `body` with `b` bound to the value of `a`, except that the element at position `(`i_1`,..,`i_k`)` is updated to the value of `v`. If `a[`i_1`,..,`i_k`]` is itself an array, i.e., if `a` has more than `k` dimensions, `v` must be an array of the same size as the slice it is replacing. We write `let a[`i_1`,..,`i_k`] = v in body` when the source and destination array share the same name. Section 2.2 describes our method for doing array updates *in-place* without losing referential transparency. This allows a cost model in which the update takes time proportional to the total size of `v`, rather than `a`.

The other non-standard construct is the `do`-loop: It is essentially syntactic sugar for a certain form of tail-recursive function, and is used by the user to express certain sequential computation that would be awkward to write functionally, and by the compiler in lower-level optimisations, e.g., loop interchange, distribution. For example, denoting by `t` the type of `x`, the loop in Figure 2 has the semantics of a call to the tail-recursive function on the right side.

2.1 Second-order array combinators

Most array operations in $\mathcal{L}_0$ are achieved through built-in second-order array combinators (SOACs). The available SOACs can be seen in Figure 3, along with our syntax for anonymous and curried functions. As $\mathcal{L}_0$ is first-order, these anonymous functions are only syntactically permitted in SOAC invocations. The parentheses may be omitted when currying if no curried arguments are given.

The semantics of the SOACs is identical to the similarly-named higher-order functions found in many functional languages, but we reproduce it here for completeness. Note that the types given are not $\mathcal{L}_0$ types, but a Haskell-inspired notation, since the SOACs cannot be typed in $\mathcal{L}_0$ itself.

$$\texttt{map}(f,a) :: (\alpha \to \beta) \to [\alpha] \to [\beta]$$
$$\equiv \{f(a\texttt{[0]}),\ \ldots,\ f(a\texttt{[n]})\}$$

$$\texttt{filter} :: (\alpha \to \texttt{bool}) \to [\alpha] \to [\alpha]$$
$$\texttt{filter}(f,\texttt{a}) \equiv \{a\texttt{[i]} \mid f(a\texttt{[i]}) = \texttt{True}\}$$

$$\texttt{reduce} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to \alpha$$
$$\texttt{reduce}(f,x,a) \equiv f(\ldots(f(f(x,a\texttt{[0]}),\ a\texttt{[1]})\ldots),\ a\texttt{[n]})$$

$$\texttt{scan} :: (\alpha \to \alpha \to \alpha) \to \alpha \to [\alpha] \to [\alpha]$$
$$\texttt{scan}(f,x,\texttt{a}) \equiv \{f(x,a\texttt{[0]}),\ f(f(x,a\texttt{[0]}),v\texttt{[1]}),\ldots\}$$

t	::=	`*[`t`]`	(Unique array)

Figure 4. Uniqueness attributes

In particular, note that `scan` and `reduce` require binary associative operators and `scan` is an inclusive prefix scan. `redomap` is a special case – it is not part of the external $\mathcal{L}_0$ language, but used internally for fusing `reduce` and `map`. Its semantics is as follows.

$$\texttt{redomap} :: (\alpha \to \alpha \to \alpha) \to (\alpha \to \beta \to \alpha)$$
$$\to \alpha \to [\beta] \to \alpha$$
$$\texttt{redomap}(\odot,g,x,v) \equiv \texttt{foldl}(g,\ x,\ v)$$

Note that the runtime semantics is a left-fold, not a normal $\mathcal{L}_0$ `reduce`. We use a Haskell-like syntax to explain the rationale behind `redomap`: `(red` $\odot$ `e) . (map f)` can be formally transformed, via the list homomorphism (LH) promotion lemma [7], to an equivalent form:
`red` $\odot$ `e . map f` $\equiv$ `red` $\odot$ `e . map (red` $\odot$ `e . map f) . split`$_p$
where the original list is distributed to p parallel processors, each of which execute the original map-reduce computation sequentially and, at the end, reduce in parallel the per-processor result. Hence the *inner* map-reduce can be rewritten as a left-fold:
`red` $\odot$ `e . map f` $\equiv$ `red` $\odot$ `e . map (foldl g e) . split`$_p$
It follows that in order to be generate parallel code for `(red` $\odot$ `e) . (map f)` we need to record either $\odot$ and `f`, or $\odot$ and `g`. We chose the latter, i.e., `redomap(`$\odot$`, g, e)`, because it allows a richer compositional algebra for fusion. (In particular, it allows to fuse `reduce` $\circ$ `map` $\circ$ `filter` into a `redomap` without duplicating computation, see Figure 14 in Section 10.)

2.1.1 Tuple shimming

As a notational convenience, $\mathcal{L}_0$ will automatically unwrap tuples passed to functions in SOACs. Precisely, if a function expects arguments $(t_1, \ldots, t_n)$, and is called with a single argument of type $(t_1, \ldots, t_n)$ (that is, a tuple containing the exact same types as expected by the function), $\mathcal{L}_0$ will automatically rewrite the function to expect a tuple, and insert the code necessary to extract the components. This permits us to write `map(op +, zip(xs,ys))`, rather than the following more cumbersome code.

```
map(fn int ((int, int) a) => let (x,y) = a in x+y,
    zip(xs,ys))
```

We will make use of this shortcut in this paper.

2.2 Safe in-place updates

When writing sequential loops, it is often very convenient to update the elements of an array in-place. However, in order to perform such an update without violating referential transparency, we must be able to guarantee that no other array that is used on any execution path following the update shares data with the updated array. To perform this check, $\mathcal{L}_0$ uses an extension to the type system in the form of *uniqueness attributes* inspired by Clean [4] and Rust [20], as well as aliasing analysis. We extend the syntax for array types to permit a prefix asterisk, as in Figure 4, denoting a unique array.

If a type is of the form `*t`, we say that it is a *unique type*. The semantics of uniqueness attributes are as follows. Inside a function, a parameter having type $*\alpha$ means that neither the argument value nor any of its aliases are going to be used after the function returns, implying that the function body can modify the argument freely. Furthermore, if the return type is unique, its value must not alias any non-unique arguments. The intuition is that the function can be considered to have exclusive access to its unique argument, and a caller to have exclusive access to a unique return value.

In a function call, a parameter having type $*\beta$ means that whatever argument is passed must be modifiable (that is, it must not be aliased with a non-modifiable function argument), and neither it nor any of its aliases may be used in any way after the function call. We say that it has been *consumed*.

As a concrete example, a function using sequential loops and inplace modification to compute the LU-factorisation of an array can be written like this. Note that the two inner loops could be written as `map`s, but have been left sequential for expository purposes.

```
fun (*[[real]], *[[real]]) lu_inplace(*[[real]] a) =
  let n = size(a) in
  loop ((a,l,u) =
    (a, replicate(n,replicate(n,0.0)),
        replicate(n,replicate(n,0.0)))) = for k < n do
      let u[k,k] = a[k,k] in
      loop ((l,u)) = for i < n-k do
        let l[i+k,k] = a[i+k,k]/u[k,k] in
        let u[k,i+k] = a[k,i+k] in
        (l,u) in
      loop (a) = for i < n-k do
        loop (a) = for j < n-k do
          let a[i+k,j+k] =
            a[i+k,j+k] - l[i+k,k] * u[k,j+k] in
          a in
        a in
      (a,l,u) in
  (l,u)
```

After an array has been used on the right-hand side of a let-with, we mark it and all of its aliases as consumed, and it may not be used afterwards. Our aliasing analysis is rather conservative. In particular, we assume that if a function returns a non-unique array, then that array may alias any of the function arguments. We also do not detect aliasing at a finer granularity than whole arrays, i.e., after `let a = b[0]`, `a` aliases all of `b`, not only its first row.

2.3 Compiler pipeline

The compilation pipeline in the current $\mathcal{L}_0$ compiler is outlined in Figure 5. Type checking is done on the original program, to ensure that any error messages refer to the names written by the programmer, but all subsequent stages consume and produce programs in which names are distinct. To begin with, we run a transformation that converts most uses of tuples to a simpler form, described in more detail in Section 2.3.1. After this comes let- and tuple-normalisation, where the program is transformed in such a way that the only direct operands to functions, SOACs and operators are variables, and that every tuple-pattern is fully expanded to cover all elements of the tuple value to be matched. One notable property of the resulting program is that no variable is ever bound to a tuple.

The enabling optimisations loop consists of:

1. aggressive inlining, i.e., building the program call-graph and inlining leaves to a fix point (inlines all non-recursive functions),
2. performing copy/constant propagation and constant folding,
3. dead code and function elimination,

which is repeated until a fixed point is reached. We also plan to add common-subexpression elimination and loop hoisting. After loop fusion, the enabling optimisations stage is repeated, as fusion will often produce code amenable to copy-propagation. By the time the program enters the enabling optimisations loop, and for the rest of the compilation pipeline, the following program properties hold:

- No tuple type can appear in an array or tuple type, i.e., flat tuples,
- `unzip` has been eliminated, `zip` has been replaced with `assertZip`, which verifies either statically or at runtime that the outer size of `zip`'s input matches, and finally, the original SOACs (`map`) have been replaced with their tuple-of-array version (`map2`, see Section 2.3.1),
- tuple expressions can appear only as the final result of a function, SOAC, or `if` expression, and similarly for the tuple pattern of a let binding, e.g., a formal argument cannot be a tuple,
- e_1 cannot be a `let` expression when used in `let` p `=` e_1 `in` e_2,
- each `if` is bound to a corresponding `let` expression, and an `if`'s condition cannot be in itself an `if` expression, e.g., `a + if(` `if` c_1 `then` e_1 `else` e_2 `)` `then` e_3 `else` $e_4 \rightarrow$ `let` c_2 `=` `if` c_1 `then` e_1 `else` e_2 `in` `let b =` `if` c_2 `then` e_3 `else` e_4 `in a+b`
- function calls, including SOACs, have their own let binding, e.g., `reduce2(f,a) + x` $\Rightarrow$ `let y = reduce2(f,e,a) in y+x`,
- all actual arguments are vars, e.g., `f(a+b)`$\Rightarrow$`let x=a+b in f(x)`.

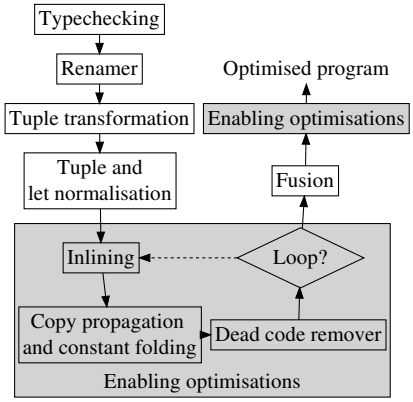

Figure 5. Compiler pipeline

The first three properties are ensured by the tuple transformation step, while the latter three are due to normalisation.

2.3.1 Tuple transformation

As mentioned above, the tuple-transformation stage flattens all tuples (i.e, `(x,(y,z))` becomes `(x,y,z)`), and converts arrays of tuples to tuples of arrays. This transformation was first developed in the context of NESL [11]. Arrays of tuples are in a sense merely syntactic sugar for tuples of arrays; the type `[(int, real)]` is transformed to `([int], [real])` during the compilation process, and all code interacting with arrays of tuples is likewise transformed. In most cases, this is fully transparent, but there are edge cases where the transformation is not an isomorphism.

Consider the type `[([int], [real])]`, which is transformed to `([[int]], [[real]])`. These two types are not isomorphic, as the latter has more stringent demands to the regularity of arrays. For example, `{({1}, {1.0}), ({2,3}, {2.0})}` is a value of the former, but the first element of the corresponding transformed tuple `({{1}, {2, 3}}, {{1.0}, {2.0}})` is not a regular array. Hence, when determining whether a program generates regular arrays, we must look at the *transformed* values - in a sense, the regularity requirement "transcends" the tuples. Also, after tuple-transformation, `zip` and `unzip` disappear.

After tuple transformation, the previously described SOACs are no longer usable, as they each accept only a single input array. Hence, we introduce matching tuple-SOACs, which accept as input an arbitrary number of arrays, and likewise their result is a tuple.

$$
\begin{array}{rcl}
e & ::= & \texttt{map2}(l,\ e_1,\ \ldots,\ e_n) \\
& | & \texttt{filter2}(l,\ e_1,\ \ldots,\ e_n) \\
& | & \texttt{reduce2}(l,\ x_1,\ \ldots,\ x_n,\ e_1,\ \ldots,\ e_n) \\
& | & \texttt{scan2}(l,\ x_1,\ \ldots,\ x_n,\ e_1,\ \ldots,\ e_n) \\
& | & \texttt{redomap2}(l_r,\ l_m,\ x_1,\ \ldots,\ x_n,\ e_1,\ \ldots,\ e_n)
\end{array}
$$

Figure 6. Second-order tuple-array combinators

Their syntax is depicted in Figure 6, and their semantics is:

$$\texttt{map2} :: (\alpha_1 \to \ldots \to \alpha_n) \to (\beta_1, \ldots, \beta_m) \to [\alpha_1] \to \ldots \to [\alpha_n] \to ([\beta_1], \ldots, [\beta_m]))$$

$$\texttt{map2}(f, e_1, \ldots, e_n) \equiv \texttt{unzip}(\texttt{map}(f',\ \texttt{zip}(e_1,\ \ldots,\ e_n)))$$

$$\texttt{reduce2} :: (\alpha_1 \to \ldots \to \alpha_n \to \alpha_1 \to \ldots \to \alpha_n \to (\alpha_1, \ldots \alpha_n)) \to (\alpha_1, \ldots, \alpha_n) \to [\alpha_1] \to \ldots \to [\alpha_n] \to ([\alpha_1], \ldots, [\alpha_n])$$

$$\texttt{reduce2}(f, x, e_1, \ldots, e_n) \equiv \texttt{reduce}(f', x, \texttt{zip}(e_1, \ldots, e_n))$$

where f' has the tuple-transformed body of f and the arguments of f have been rewritten such that an original argument of tuple type is expanded into distinct arguments. The remaining SOACs are similar. For simplicity, the above semantics always describe the return value as a tuple. In practice, if this would be a one-element tuple (which is not permitted in $\mathcal{L}_0$), we use the element by itself. The array-arguments to a tuple-SOAC must all have the same length.

As an example, consider the following (contrived) program for computing the dot product of an array and its inverse.

```
fun real main([real] a) =
  reduce(op +, 0.0,
         map(op *,
             map(fn (real,real) (real x) => (x,~x),
                 a)))
```

Now we perform the tuple-transformation. Note that the return value of the first `map2` is taken apart in a tuple pattern, so that it can be passed piecewise to the next `map2`.

```
fun real main([real] a) =
  let (e1, e2) =
    map2(fn (real, real) (real x) =>
          (x,~x), a) in
  let tmp_map2 =
    map2(fn real (real x, real y) => x * y,
          e1, e2) in
  let tmp_red2 =
    reduce2(fn real (real x, real y) =>
              x + y, 0.0, tmp_map2) in
  tmp_red2
```

3. Fusion: A Structural-Analysis Transformation

The fusion transformation, presented in this section, assumes a *normalized* program, i.e., by running the transformations introduced in Section 2 to obtain the properties listed in Section 2.3

Structural analysis is rooted in the T_1-T_2 transformation [1] depicted in Figure 7. If repeated application of T_1 and T_2 to a control-flow graph (CFG) results in one point, then the CFG is said to be reducible, i.e., the code can be re-written using only regular (while) `loops`, `if` and `goto`-free statements (with function calls).

Data-flow optimizations on reducible CFGs can be modeled via equations that are applied at each T_1/T_2 reduction, and consequently only one CFG pass is required instead of a fixed-point iteration. In practice, if the CFG is known to be reducible, then analysis can be conveniently performed source to source: data-flow equations are associated directly with the language constructs and dictate, for example, how the analysis result is initialized at statement

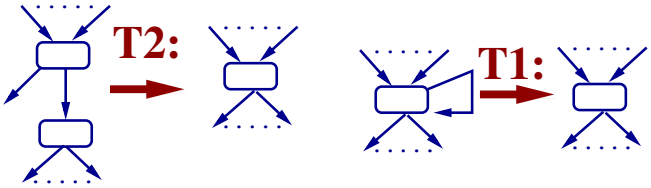

Figure 7. T_1-T_2 Transformation For Testing CFG Reducibility

level, composed between consecutive statements, merged across branches, aggregated across loops, and translated across call sites. (An example of such non-trivial analysis is the summarization [29] of array references into read-write, write-first and read-only set expressions, used in the context of array-SSA and autoparallelization.)

Since $\mathcal{L}_0$ guarantees a reducible CFG, fusion is implemented as an intra-procedural[1] source-to-source transformation: a bottom-up traversal of the abstract-syntax tree (AbSyn) builds the "fusion kernels" and a second pass substitutes them in the AbSyn and cleans up the code. Such an approach is not uncommon.

What is less common is that the data-flow equations themselves model T_2-like reducibility of the data-dependency graph. The remaining of this section is organized as follows: Section 3.1 gives the gist of the technique, and shows several don't-fuse cases, which would either lead to illegal programs or to duplicated computation. Section 3.2 presents the data structures and data-flow rules of the first analysis pass for several of the $\mathcal{L}_0$ constructs. Section 3.3 discusses the central data-flow rule that merges a second-order array combinator (SOAC) to the fusion result. Finally, Section 3.4 presents the compositional algebra under which SOACs are fused, and briefly discusses the second analysis pass.

3.1 Motivation and Intuitive Solution

Figure 8 depicts the intuitive idea on which our fusion analysis is based. The top-left figure shows the dependency graph of a simple program, where an arrow points from the consumer to the producer. For simplicity, array variables are used only as input or output to SOAC calls and the control flow is trivial, i.e., a basic block.

The main point is that all SOACs that appear inside the box labeled *Kernel 3* can be fused without duplicating any computation, even if several of the to-be-fused arrays are used in different SOACs, e.g., `y1` is used to compute both `(z1,z2)` and `res`[2]. This is accomplished by means of T_2 *reduction on the dependency graph*:

The rightmost child, i.e., `map2(g,..)`, of the root SOAC has only one incoming edge, hence it can be fused (reduced). This is achieved (i) by replacing in the root SOAC the child's output with the child's input arrays, (ii) by inserting in the root's lambda a call to the child's lambda, which computes the per-element output of the child, and, finally, (iii) by removing the duplicate input arrays of the resulting SOAC. The latter introduces copy statements for all but one of the (former) arguments of the lambda corresponding to the same duplicated array (and removes those former arguments).

The top-right part of Figure 8 shows in blue the (optimized) result of the first fusion, where the copy statements have been eliminated by copy propagation. In the new graph, the leftmost child of the root, i.e., the one computing `(z1,z2)`, has only one incoming edge and can be fused. The resulting graph, shown in the

[1] The current version of the $\mathcal{L}_0$ compiler relies on aggressive inlining and does not support (yet) inter-procedural analysis.

[2] Note also that (i) not all input arrays of a SOAC need to be produced by the same SOAC, e.g., the `qs` requires both `ys` and `zs` arrays, and (ii) some input might be produced other than by a SOAC, e.g, *Kernel 3* is still fusable even if we add an arbitrary array as an extra parameter to the root `res=map2(..)`.

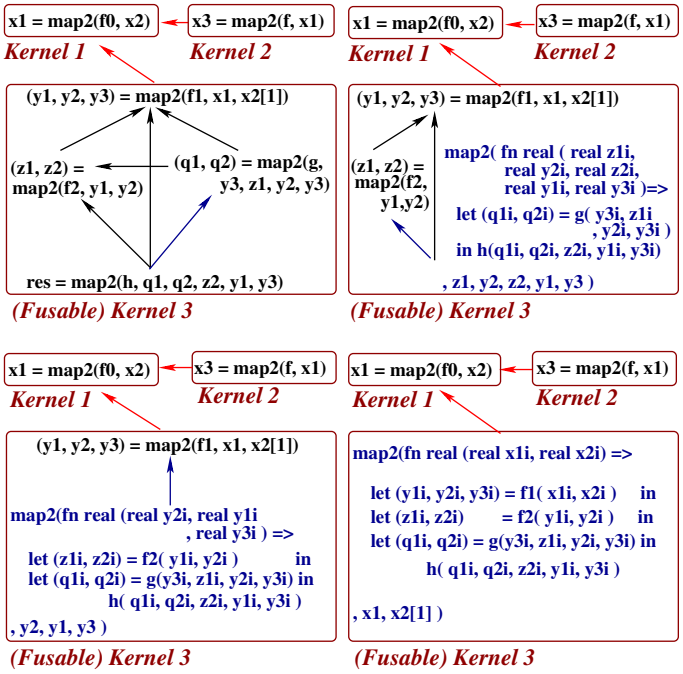

Figure 8. Fusion By T2 Transformation on the Dependency Graph

```
// Case 1: don't fuse if it
//moves an array use across
//its in-place update point
let x   = map2(f, a) in
let a[1]= 3.33       in
let y   = map2(g, x) in ..

// Case 3: don't fuse from
//outside in a loop (or λ)
let x = map2(f, a)   in
loop(arr) = for i < N do
  map2(op +, arr, x)

// Case 5: don't fuse if x
//used outside SOAC inputs
let x = map2(f, a) in
let y = map2(g, x)   in
    x[i] + y[i]
```

```
// Case 2: not all SOAC
//combinations are fusable
let x = filter2(c1, a) in
let y = filter2(c1, b) in
let z = map2   (f,x,y) in..

// Case 4: don't fuse in 2
//kernels sharing a CF path
let x = map2(f, arr)  in
let y = if c then map2(g1,x)
                  else map2(g2,x)
in let z = map2(h, x) in ..

// Case 6: x & y used exactly
//once but still don't fuse
let (x,y) = map2(f, a)   in
let u = reduce2(op +,0,x)in
let v = reduce2(op *,1,y)..
```

Figure 9. Don't Fuse Cases: Illegal or Duplicates Computation

bottom-left figure can be fused again resulting in the bottom-right graph of Figure 8. At this point no further T_2 reduction is possible because the SOAC computing `x1` has two incoming edges. *When no* T_2 *reduction is possible a new kernel is started*, e.g., *Kernel 1*.

Having presented the intuitive idea, we look next at six cases, depicted in Figure 9, where fusion is disallowed because it would result either in incorrect programs or in duplicated computation:

1. A SOAC `s` cannot be fused across the "in-place update" of an array `a` if `s` uses any variable in the alias set of `a`. Otherwise, fusion would violate $\mathcal{L}_0$'s semantics because an alias of `a` is used on an execution path following `a`'s in-place update.
2. Not all combinations of SOACs are fusable. For example, a `map` whose input arrays are produced by two `filter` operations can be fused as a loop, but this is not useful due to the sequential nature of the resulting loop, i.e., it uses two induction variables without closed-form solutions.
3. Fusing across a loop (or a SOAC's lambda) would duplicate computation and potentially change the time complexity of the program, because the loop-invariant computation of the producer would be redundantly executed loop-count times.
4. If a SOAC-produced array `x` is consumed by two other SOACs at two program points located on the same execution path p then the fused program would compute `x` twice on p. However, if the two program points are located on disjoint execution paths then fusion is allowed. For example, if the `map2` computing `z` is removed, then fusing the `x` producer in the `then` and `else` branches does not duplicate computation.
5. If a SOAC-produced array `x` is used other than input to another SOAC then fusing `x` is disallowed in our implementation. A future extension might handle the "negligible-overhead" cases, e.g., substituting `x[i]` with `f(a[i])` would allow the fusion of `x` at the cost of computing `f(arr[i])` twice.
6. Finally, fusing two arrays produced by the same SOAC each in another SOAC still duplicates computation. This case can sometimes be optimized by horizontally fusing the SOAC consumers, e.g., the two `reduce`2s are merged in `reduce2(g,0,1, x,y)`, where `g(`e_1`,`e_2`,`x_i`,`y_i`)` $\equiv$ `(`e_1`+`x_i`, `e_2`*`y_i`)`.

We conclude this section with two remarks: *First*, the data-dependency graph (DDG) does not have to be built since a bottom-up traversal of the program, i.e., backwards analysis, is guaranteed to encounter the statements[3] in an order that satisfies the DDG. *Second*, the intuition is not complete, as it does not solve the issues described in the "don't fuse" cases, e.g., violation of the "in-place update" semantics, handling fusion across loops and branches, etc. The next two sections present in detail the backward-analysis pass.

3.2 Data Structures and Data-Flow Rules for Fusion

The $\mathcal{L}_0$ compiler is implemented in Haskell. Figure 10 shows the structure of the data-flow result of the bottom-up analysis pass, i.e., a synthesized attribute. A fused kernel, `FusedKer`, consists of:

- the SOAC statement, `soacStmt`, which pairs up the output arrays of a fused SOAC with its SOAC's (ABSYN) expression,
- the set of input arrays, `inp`, of the fused SOAC,
- a set of variable names, `inplace`, which is the union of the alias sets of all arrays that may have been "consumed[4]" on any execution path between the currently-analyzed program point to the one where the fused SOAC was called; e.g., in Case 1 of Figure 9, `a` belongs to the `inplace` set of the kernel associated with output `y`, when analysis reaches the definition of `x`,
- a set of array variables, `fused_vars`, that have been fused in the construction of the current kernel; if `fused_vars` $= \emptyset$ then no fusion has taken place, and `soacStmt` remains in the program.

The analysis result is implemented by the `FusionRes` structure:

- `outArr` maps an array name to the kernel (name) producing it,
- `inpArr` maps an array name `x` to the set of kernels (names) whose corresponding SOACs receive `x` as an input array,
- a set of array names that the analysis up to the current program point have discovered to be `unfusable`, because of one of the cases 3 to 6 in Figure 9, e.g., arrays used other than SOAC input or in two different kernels that share an execution path, etc., and
- `kers` maps a kernel name to its associated `FusedKer` data.

[3] Fusion assumes a normalized program, i.e., seen as a set of functions whose bodies are composed from blocks of let-statements, `if`s and `loop`s.

[4] An array is consumed either (i) when it is the source of an in-place update or (ii) when it is passed to a function call as a parameter of unique type.

```
import qualified
  Data.Map as M
import qualified
  Data.Set as S
data FusedKer = FusedKer{
 soacStmt :: ([Name],Exp)
--^the fused SOAC stmt,eg,
--(z,w)=map2( f(a,b), x,y)

,inp     :: S.Set Name
--^the input arrays used
--in SOAC stmt,i.e.,{x,y}

,inplace :: S.Set Name
--^Aliasing set of vars
--used in in-place updates
--that reach this kernel.

,fused_vars :: [Name]
--^not null iff at least
--a fusion was performed
}
```

```
data FusionRes = FusionRes{
 outArr :: M.Map Name Name
--^maps an array to the name
--of the kernel producing it

,inpArr :: M.Map  Name
                  (S.Set Name)
--^maps an array to the
--names of kernels using it

,unfusable :: S.Set Name
--^Unfusable arrays. Used:
--1.otherwise than input to
--  SOAC kernels (including
--  lambda bodies), or
--2.as input to two kernels
--  not located on disjoint
--  control-flow (branches)

,kers ::M.Map Name FusedKer
--^maps kernel name to data
}
```

```
data FusionEnv = FusionEnv {
    soacsEnv :: M.Map Name ([Name], Exp)
    --^ maps array names to their producing SOAC stmt
  , varsEnv  :: S.Set Name
    --^ set of in-scope variables at current prog point
}
```

Figure 10. Data Structures for Fusion's Result and Environment

```
allOutArrs :: [Name] -> FusionM [Name]
allOutArrs =
-- allOutArrs [x] → [x,y]  if let (x,y) = map2(f,a)
  foldM (\y nm -> do bnd <- asks $ M.lookup nm . soacsEnv
                     case bnd of
                       Nothing    -> return (nm:y)
                       Just (s,_) -> return (s++y)
        ) [ ]
composeRes :: FusionRes -> FusionRes -> FusionM FusionRes
composeRes res1 res2 = do
  let ufus = unfusable res1 `S.union` unfusable res2
  inp_arr1 <- (allOutArrs . M.keys . inpArr) res1
  inp_arr2 <- (allOutArrs . M.keys . inpArr) res2
  return $ FusedRes
    ( outArr  res1  `M.union`  outArr  res2)
    ( M.unionWith S.union (inpArr res1) (inpArr res2) )
    ( ufus `S.union` (inp_arr1 `S.intersection` inp_arr2))
    ( kernels res1  `M.union`  kernels res2)

unionRes :: FusionRes -> FusionRes -> FusionM FusionRes
unionRes res1 res2 = ... -- semantically unions results:
--same as composeRes except that the resulting unfusable
--set is ufus (the union of the two input unfusable sets)
```

Figure 11. Composing two regions on the same and disjoint paths.

We note that `inpArr` ∪ `unfusable` covers all the array variables defined in the program, except for the ones that are currently out of scope. The analysis uses an environment, `FusionEnv`, that records the set of array variables visible at the current program point, `varsEnv`, and a map that binds an array name to the SOAC statement producing it, `soacsEnv`. The environment is computed during the top-down AbSyn traversal, i.e., an inherited attribute. The computation takes place in the `FusionM` monad that makes the `FusionEnv` environment available via a `Reader` monad interface.

Figure 11 shows the helper functions `composeRes` and `unionRes` that implement the data-flow equations for merging the results of

```
fusion1 :: FusionRes -> Exp -> FusionM FusionRes
fusion1 r (Var id0) =
  if not $ isArrayType id0 then return r
  else let ufus' = S.insert id0 (unfusable r)
       in return r { unfusable = ufus' }

fusion1 r (let id1 = id0 with
           [ inds ] <- elem in body) = do
    r'  <- bindVarsEnv [id1] $ fusion1 r body
    -- Add the aliases set of id0 (included)
    -- to the 'inplace' field of any kernel:
    let kers = M.map (addToInplace id0) (kernels r')
    let r'' = r' { kernels = kers }
    foldM fusion1 r'' (elem : Var id0 : inds)

fusion1 r (if e_c then e_then else e_else) = do
    r_then <- fusion1 mkNullRes e_then
    r_else <- fusion1 mkNullRes e_else
    r_cond <- fusion1 r         e_c
    return $ composeRes r_cond (unionRes r_then r_else)

fusion1 r (fn (ids) => body) = do
    r' <- bindVarsEnv ids $ fusion1 mkNullRes body
    return $ composeRes r (composeRes r' r')

fusion1 r (let pat = rhs in body) =
  case rhs of
    map2(fun, inputs) -> do
      r' <- bindBothEnvs pat rhs $ fusion1 r bdy
      r'' <- fusion1 r' lam
      tryFuseSOAC (freeInBody fun) r'' (pat, rhs)
      -- similar for reduce2, filter2, redomap2.
      -- Replicate has a specialized implem (not shown)
    _ -> do r' <- bindVarsEnv pat $ fusion1 r body
            fusion1 r' e
```

Figure 12. Constructing Fused Kernels at Function Level

two code regions on the same and disjoint control-flow paths, respectively. `unionRes` performs the (semantic) union of the results of the two branches, e.g., the union of the `unfusable` sets and the union of the fusion kernels, etc, because a SOAC can be fused in two kernels on separate branches without duplicating computation. `composeRes` is similar to `unionRes`, except that the intersection between the `inpArr` key sets becomes `unfusable`, i.e., because the arrays in the intersection may be used by two kernels on the same execution path, and hence further fusion of their producing SOACs may duplicate computation.

Figure 12 summarizes the most-relevant data-flow rules of the first analysis pass. Function `fusion1` provides the implementation, where the arguments represent the current data-flow result, denoted `r`, and an expression. If the expression is:

- an array variable, `Var idd`, then the variable's name is added to the unfusable set of the result, since it corresponds to an array used other than as input to a SOAC, i.e., Case 5 in Figure 9,
- an in-place update expression then (i) the new binding is added to the `varsEnv` set of in-scope variables, (ii) the `inplace` field of each kernel in the current result is updated with the alias set of the consumed variable, and (iii) the contribution of sub-expressions is added to the data-flow result. A function call that "consumes" some of its parameters is treated similarly. This solves Case 1 of Figure 9.
- an if-then-else then the data-flow result of the `then` and `else` branches are computed independently starting from a (fresh) null result, because no fusion is possible either across branches or with the kernels obtained from the expression following the `if` (visibility issues). The results, i.e., $\texttt{then}_r$ and $\texttt{else}_r$,

correspond to disjoint paths and are composed via `unionRes`. Finally, the original result `r` is updated with the contribution of the `if` condition, yielding $\texttt{cond}_r$, and the overall result is computed via `composeRes` since it corresponds to overlapping control-flow paths. This solves both issues of Case 4 in Figure 9.

- a lambda then the result for the lambda's body, `r'`, is computed from a null result because any of the arrays defined inside the lambda are invisible outside. Since a lambda's body is executed multiple times inside a SOAC then the data flow equation is semantically: `composeRes r (composeRes r' r')`, i.e., the code regions corresponding to `r` and `r'` share an execution path. In particular, all variables used in the lambda's body become `unfusable`, which prevents fusion in a lambda (or loop) from outside it, i.e., Case 3 in Figure 9. The actual implementation performs less computation and also filters out the variables that are invisible in the outer scope. A loop is treated similarly.
- the let-binding of a `map2`, `reduce2`, `redomap2` or `filter2` statement, then (i) both environment variables, i.e., `soacsEnv` and `varsEnv`, are updated with the new bindings, (ii) the body of the SOAC lambda is processed, e.g., the variables used inside lambda become unfusable, and (iii) `tryFuseSOAC` (see next Section 3.3) either fuses the SOAC or creates a new kernel.
- an arbitrary let-binding then the `varsEnv` environment variable is updated with the new binding and the sub-expressions, i.e., `body` and `e`, are processed recursively. `scan2` is processed in a similar fashion since it is not part of the fusion algebra.

3.3 Fusing One SOAC

Figure 13 shows the Haskell pseudo-code that implements the central step of analysis: the data-flow rules for processing a SOAC statement, `tryFuseSOAC`. Its parameters are: (i) the set of variables that are visible in the current scope and are used in the current-SOAC's lambda, `lam_vars`, (ii) the current data-flow result, `res`, and (iii) the output arrays and the expression of the to-be-processed SOAC statement, `(out_nms,soac)`.

There are four conditions, `all4_ok`, that have to be met for the current SOAC, denoted S_c, to be fused with at least one kernel:

1. None of S_c's output-array names, `out_nms`, belong to the result's `unfusable` set.
2. There exists some kernels, `to_fuse_kers`, found by look-ups in the result's `inpArr`, whose input arrays belong to `out_nms`. If both conditions are met then it is guaranteed (see Section 3.2) that all `to_fuse_kers` are located on disjoint execution paths, and that none of them are in a loop or lambda that does not contain S_c, i.e., Cases 2 to 6 of Figure 9 do not happen.
3. All kernels in `to_fuse_kers` are compatible with S_c under the algebra depicted in Figure 14 of Section 3.4. Otherwise, if only some kernels are compatible, then computation will be duplicated since S_c cannot be removed from the program.
4. None of S_c's variables, including the ones used in its lambda and as input arrays, belong to the `inplace` set of any kernels in `to_fuse_kers`. Otherwise, Case 1 of Figure 9 may apply.

The next step is to update the `unfusable` set of the data-flow result. At this stage the variables used in the current SOAC (S_c) lambda have been already made unfusable. It remains to check whether any input array[5] of S_c, i.e., `inp_nms`, is also used in an existing kernel. If so, then it is used in at least two kernels that may share an execution path, and hence it is unfusable. We make two remarks: *First*, if an array in the input set of S_c was produced by another SOAC, S_2, then `inp_nms` is extended with all output arrays of S_2; otherwise Case 6 of Figure 9 may apply. This is achieved by `allOutArrs` (shown in Figure 11) via looks-ups in `soacsEnv`.

Second, an input array, `x`, does *not* become unfusable if S_c can be fused *and* `x` is used only as input to the kernels with which S_c will be fused. In this case `x` would still be used only in kernels located on disjoint execution paths (This is implemented by filtering modulo kernels `mod_kers` in the definition of `in2_kers`.)

Finally, if any of the four fusion conditions are not met, then a new kernel is created; otherwise the current SOAC is fused with each of the kernels in `to_fuse_kers`.

[5] A normalized program guarantees that the input arrays of a SOAC are variables rather than arbitrary expressions, hence we need not scan them.

```
tryFuseSOAC:: S.Set Name ->            FusionRes ->
              -- vars ∈ SOAC's lamba,   current result
               ([Name], Exp) -> FusionM FusionRes
              -- SOAC stmt,     result after fusion
tryFuseSOAC lam_vars res (out_nms, soac) = do
  inp_ids <- getInpArrsSOAC soac
  -- e.g., [x,y] ⇐ map2(f, x, y)

  -- Conditions for fusion:
  --(1) none of out_nms belongs to the unfusable set
  let cond1=not $ any (`S.member` unfusable res) out_nms

  --(2) ∃ some kernels that use some of out_nms as inputs
  let to_fuse_knms = getKersWithInpArrs res out_nms
  let to_fuse_kers =
    fromJust $ mapM (`M.lookup` kers res) to_fuse_knms
  let cond2 = not $ null to_fuse_kers

  --(3) all kernels have to be compatible for fusion,
  --    e.g., map2 o filter2 not supported
  let cond3 =
    all (isCompatibleKer out_nms soac) to_fuse_kers

  --(4) fusion cannot move a use of an input array
  --    past its in-place update
  let used =
    S.intersection $ lam_vars `S.union` S.fromList inp_ids
  let cond4 = all (S.null . used . inplace) to_fuse_kers

  let is_fusable = cond1 && cond4 && cond3 && cond4

  -- Update Unfusable Set with the input-array names that
  -- appear as input arrays in kernels ∉ to_fuse_kers,
  -- since those are input to at least 2 distinct kernels.
  inp_nms <- allOutArrs inp_ids
  -- e.g., [x,y] ⇐ map2(g, x)  if let (x,y) = map2(f,a)
  let mod_kers=if is_fusable then to_fuse_knms else []
  let in2_kers=filter (inpArrInRes res mod_kers) inp_nms
  let res' = res { unfusable = unfusable res `S.union`
                               S.fromList in2_kers }

  if is_fusable
  then ... -- fuse current soac with all to_fuse_kers
           -- update out/inpArr & kers fields of result
  else ... -- add a fresh kernel to the result, and
           -- update the outArr and inpArr fields
```

Figure 13. Pseudo-code for Conservatively Fusing one SOAC

3.4 Fusion's Composition Rules and Second Analysis Pass

The algebra under which fusion is performed is depicted in Figure 14: `scan2` is unfusable and `reduce2` and `redomap2` always start a new kernel. Since `replicate` is semantically a `map2` with a constant function, it can always be fused without duplicating computation, even inside loops and SOAC's lambdas, except for the cases when it violates the in-place semantics, i.e., Case 1 of Figure 9. If the current SOAC, S_c, is a `map2` then it can be fused with a `map2`, `reduce2`, or `redomap2` kernel, and produces a `map2`,

```
// replicate can be fused
// without restrictions
let x = replicate(N,a)in
let y = map2(f, x, b) in
let z = map2(g, x, c) in
let x[i] = ...
    ≡
let x = replicate(N, a) in
let y = map2( fn β1 (α1 bi)
              => f(a,bi), b)
let z = map2( fn β2 (α2 ci)
              => g(a,ci), c)
in let x[i] = ...


//map2 o map2 ⇒ map2
let (x1, x2) = map2(f, a1)
in  map2(g, x1, y)
    ≡
map2(fn β (α1 a1i, α2 yi)
  =>let (x1i, x2i) = f(a1i)
    in  g(x1i, yi)
, a1, y )


//reduce2 o map2⇒redomap2
let (x1, x2) = map2(f, a1)
in  reduce2(⊕,e1,e2, x1,y)
    ≡
redomap2(⊕
, fn (β1,β2) ( β1 e1, β2 e2
             , α1 a1i,α2 yi)
   => let (x1i, x2i) = f(a1i)
      in  ⊕(e1,e2,x1i,yi)
, (e1, e2), a1, y )


//redomap2 o map2⇒redomap2
let (x1, x2) = map2(f, a1)
in  redomap2(⊕, g, e, x1, y)
    ≡
redomap2(⊕
, fn β (β e, α1 a1i, α2 yi)
   => let (x1i, x2i) = f(a1i)
      in  g(e, x1i, yi)
, e, a1, y )
```

```
//filter2 o filter2⇒filter2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c1,a1,a2)
in  let y = filter2(c2, x1) ..
    ≡
let (y, dead) = filter2(
  fn bool (α1 a1i,α2 a2i)=>
      if   c1(a1i, a2i)
      then c2(a1i)
      else false
, a1, a2 ) ..

//reduce2 o filter2⇒redomap2
//IFF consumer's input list
//  ≡ producer's output list
let x = filter2(c, a)
in  reduce2(⊕, e, x)
    ≡
reduce2(fn β (β e, β ai) =>
  if c(ai) then ⊕(e,ai) else e
, e, a )

//reduce2 o filter2⇒redomap2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c, a1, a2)
in  reduce2(⊕, e, x1)
    ≡
redomap2(⊕
, fn β (β e, α1 a1i, α2 a2i)
   => if c(a1i, a2i)
      then ⊕(e, a1i) else e
, e, a1, a2 )

//redomap2 o filter2⇒redomap2
//IFF consumer's input set
//  ⊆ producer's output set
let (x1,x2)=filter2(c, a1, a2)
in  redomap2(⊕, g, e, x1)
    ≡
redomap2(⊕
, fn β (β e, α1 a1i, α2 a2i)
   => if c(a1i, a2i)
      then g(e, a1i) else e
, e, a1, a2 )
```

Figure 14. Compositional Algebra For Fusion

redomap2 and redomap2 kernel, respectively. Note that s_c does not have to produce all of the kernel's input arrays.

If s_c is a filter2 then, with the current algebra, it can be fused with a filter2, reduce2 or redomap2 kernel, but only when s_c's result-array set is a superset of the kernel input-array set. (Note that when the input and output arguments match, reduce2 o filter2 results in reduce2, rather than redomap2.) In the case of a filter2 kernel, the output-array set might need to be extended with fresh (dead) variables, and the order of the input/output arrays might need adjustment to satisfy the typing rule of filter2.

One can observe that redomap2, which is not part of the user-visible language, is instrumental in enhancing the composibility degree of the fusion algebra, while preserving the parallel semantics of the result. For example, a reduce2 can be fused with a filter2, then with several (partial) map2s, then again with a filter2, etc.

The result of the first (bottom-up) analysis pass has thus been summarized at each function level. The next step is to filter out the kernels that have not been fused, i.e., fused_vars= $\emptyset$, from the result. The second analysis pass then replaces in the program the SOAC statements whose output arrays are keys in outArr with the fused SOAC of their associated kernel. Since successful fusion may have created opportunities for fusion at an inner level, each lambda corresponding to a fused SOAC is (i) first cleaned-up by running the enabling optimizations, and (ii) then the two passes are re-run on the lambda's body. Finally, at the very end, the enabling optimizations are applied to clean up the whole program, e.g., dead-code elimination removes the SOACs that have been already fused.

4. Discussion, Possible Extensions, and Statistics

This section is structured as follows: Section 4.1 discusses how to solve certain cases where fusion is inhibited due to calls to some of the built in functions such as size, split, etc. Section 4.2 presents two code transformations that are aimed at optimizing uses of scan2 and reduce2. Since these transformations both enable and are enabled by fusion, we plan to encompass them in our fusion analysis implementation in the near future. Section 4.3 shows fusion-related statistics from six benchmarks, which were gathered by compiler instrumentation.

4.1 Fusion Hindrances

We have seen that the structural analysis presented in the previous section may allow fusion even when a variable is used as (an) input (array) to several second-order array combinators (SOAC), such as map2, reduce2. However, often enough, fusion is impeded by a SOAC input array being used as argument to built-in functions such as split, transpose, size, assertZips[6], etc. For example,

```
let x = map2(f, a)       in    let n = size(a)/2       in
let n = size(x)/2        in    let (a1,a2)=split(n,a)  in
let (x1,x2)=split(n,x)   in ⇒  let x1 = map2(f, a1)    in
let y = reduce2(g,x1,x2)..     let x2 = map2(f, a2)    in
                               let y = reduce2(g,x1,x2)..
```

the use of x in size and split in the lefthand-side code would inhibit fusion. The code on the right side suggests that a possible solution is to propagate the inhibitors as far up in the program as possible, e.g., a and x are the input and result of a map2, therefore they must have the same (outermost) size. Now x_1 and x_2 can be fused inside the map2 producer of y.

To overcome such cases, we have extended our analysis to associate call statements of inhibitor functions (for now size and assertZip) with the kernels that use them, for example, the argument of size as input, and by performing the necessary translations, e.g., size(x) ⇒ size(a), at the time when x is fused.

Figure 15 demonstrates the application of our analysis to a matrix multiplication implementation [22] that uses flat-parallelism. One can observe that (i) the obtained code resembles the common implementation, i.e., a reduce ∘ map inside two nested maps, (ii) that all replicates have been eliminated by fusion, and (iii) that the assertZips have been moved as to not hinder fusion.

4.2 Possible Extensions: ISWIM/IRWIM and REDFLAT

The fusion algebra for reduce2 and scan2 is relatively poor, for example scan2 is not fused at all, while reduce2 just starts a new kernel that, under fusion, becomes redomap2. This section presents three high-level transformations, named ISWIM, IRWIM and REDFLAT, that have scan2 and reduce2 as principal actors, and may either enable fusion or be enabled by fusion or both.

The top part of Figure 16 shows the intuitive idea behind ISWIM: a scan operation on a matrix in which the binary associative operator is (zipWith ⊙) has the same semantics as transposing the matrix, mapping each of the rows, i.e., former columns,

[6] assertZip checks that all (array) arguments have the same outer size and is produced by the array-of-tuple transformation when a zip is eliminated.

```
// FLAT-PARALLELISM MATRIX MULTIPLICATION
fun   int   redplus1( [int]  a) = reduce(op +, 0, a)
fun  [int]  redplus2([[int]] a) = map   (redplus1, a)

fun  [int]  mul1( [int]  a, [int]  b)=map(op *,zip(a,b))
fun [[int]] mul2([[int]] a,[[int]] b)=map(mul1,zip(a,b))
fun [[int]] replin(int N,   [int]  a)=replicate(N, a)

fun [[int]] matmultFun(int N, [[int]] x, [[int]] y ) =
    let yt  = replicate( N, transpose(b) ) in
    let ar  = map      ( replin(N),    x ) in
    let abr = map      (mul2, zip(ar, yt)) in
        map(redplus2, abr)

// MATRIX MULTIPLICATION AFTER FUSION
fun [[int]] matmultFun(int N, [[int]] x, [[int]] y ) =
  let yt = transpose(y) in
  let d1 = assertZip(x, iota(N)) in
  let res=
    map2(fn [int] ([int] a) =>
           let d2 = assertZip(yt, iota(N)) in
           let t1 =
             map2(fn int ([int] t2) =>
                    let d3 = assertZip(t2, a) in
                    let t3 =
                      redomap2(fn int (int u, int v )
                                  => u + v
                              ,fn int(int e,int t4,int t5)
                                  => let p = t4 * t5
                                     in e + p
                              ,0, a, t2)
                    in t3
                  , yt)
           in t1
       , x)
  in res
```

Figure 15. Matrix Multiplication Before and After Fusion

with (scan $\odot$) and transposing back the result. A similar result can be derived for reduce.

In principal, this transformation interchanges the scan/reduce with the inner map, hence ISWIM/IRWIM, to the result that the transformed code can be executed as a segmented scan [10], i.e., exploiting both levels of parallelism, rather than choosing between the parallel scan and the parallel map. Furthermore, pushing the least parallel construct, i.e., scan, at the innermost position might reveal a deeper map-nest, e.g., if the original scan was inner to a map, thus increasing significantly the depth-one parallelism degree. Finally, if the created map nest exhibits enough parallelism, then the scan can be executed sequentially rather than in parallel.

The middle part of Figure 16 shows several useful extensions to our algebra: map2^n, defined inductively, is simply a n-level perfect nest of map2s. The transpose operator receives as arguments the dimension (number) to be transposed, k, the new position of that dimension, n, and the multidimensional array. The index-changing result is that all dimensions between k+1 and n are shifted left by 1.

The bottom part of Figure 16 shows that fusion can operate across transpose via a transformation similar to the one discussed in Section 4.1. One can observe that ISWIM is both (i) a fusion enabler, i.e., interchanging the scan with the map may create fusion opportunities, and (ii) a beneficiary from fusion, i.e., the perfect map nest may be created by fusion.

The top part of Figure 17 shows the generalization for ISWIM (IRWIM) for a depth-n map nest. The differences are (i) that an extra nth dimension is created for each neutral element, in order to match the input-array sizes for the map, and (ii) the two-way use of the generalized transpose for bringing the first dimension to position n and back again. We remark that the extra replicate can

```
// Interchanging Scan With Inner Maps (ISWIM) Example:
transpose :: [[α]] → [[α]]
b = transpose(a) ⇒ a[i1,i2] ≡ b[i2,i1]

scan2( fn [real] ([real] x, [real] y) => map2(op +, x, y),
     , {0.0,..,0.0}, a )   ≡
transpose( map2 (fn [real] ([real] x)=>scan2(op +,0.0,x)
                , transpose(a) )

// Generalization for Nested Map2 (similar for reduce2)
map2^1(f, a1,.., ak) ≡ map2(g, a1,.., ak)
map2^n(f, a1,.., ak) ≡
map2(fn ([β1],..,[βt]) ([α1] x1,..,[αk] xk) =>
        map2^{n-1}(f, x1,.., xk)
    , a1,.., ak )

// Generalization for Transpose:
transpose :: ( Int,  Int, [1[..qα]]) → [1[..qα]]
b=transpose(k,n,a) ⇒ a[i1,..,ik,ik+1,..,ik+n,..,iq] ≡
                     b[i1,..,ik+1,..,ik+n,ik,..,iq]

// Fusing Across Transpose (Similar for Reshape/Flatten):
let x=map2^n(f,a) in let y=transpose(1,n-k,x) in map2^n(g,y)
        ≡
map2^n(g o f, transpose(1,n-k,a) )
// i.e., the map2 produced by ISWIM may be further fused.
```

Figure 16. Interchange Scan With Inner Maps (ISWIM) Transform.

```
// Arbitrary-Nested-Level Generalization of ISWIM
// similarly Interchange Reduce with Inner Maps (IRWIM)
scan2( fn ( [1[..nα1]],    .., [1[..nαk]] )
          ( [1[..nα1]] x1^1, .., [1[..nαk]] xk^1,
            [1[..nα1]] x1^2, .., [1[..nαk]] xk^2 ) =>
              map2^n(⊕, x1^1,.., xk^1, x1^2,.., xk^2)
     , (ne1, ..., nek), a1, ..., ak
     )              ≡
let (.., ret, ..) = (.., map^n( replicate(1), net ), ..)
// replicate dim n of neutral elems so map2 sizes match
let ( y1,.., yk ) =
  map2 (fn ( [1[..nα1]],    .., [1[..nαk]] )
           ( [1[..nα1]] x1, .., [1[..nαk]] xk ) =>
          map2^{n-1}( fn ([1[..n-1α1]],  .., [1[..n-1αk]] )
                         ([1[..n-1α1]] e1,..,[1[..n-1αk]] ek,
                          [1[..n-1α1]] x1,..,[1[..n-1αk]] xk)
                        =>scan2(⊕, e1[0],..,ek[0], x1,..,xk)
                      , re1, ..., rek, x1, .., xk )
       , transpose(1,n,a1), .., transpose(1,n,ak) )
in (transpose(n, q1-n, y1), ..., transpose(n, qk-n, yk))
// transpose back the result; qt is the dimension of αt

// Reduce Flattening Transformation (RedFlat)
y = redomap2^n ( ⊕
                , fn (α1,..,αk) ( α1 e1,..,  αk ek,
                                  [α1] x1,..,[αk] xk ) =>
                     reduce(⊕, e1,..,ek, x1,..,xk)
                , e1,..,ek, a1,..,ak )
        ≡
reduce2(⊕, (e1,..,ek), flatten(n, a1),.., flatten(n, ak))
// flatten(n,a) flattens the first n dims of array a;
```

Figure 17. ISWIM Formalization & Reduce Flattening Transform.

be compiled away by fusion coupled with a refinement of the copy-propagation transformation (both supported by current compiler).

The second transformation, named REDFLAT and depicted in the bottom-part of Figure 17, relies on the well-known equivalence:

```
let x=map2(fn int ([int] x) =>   ≡   reduce2( op +, 0,
        reduce2(op +,0,x), a)            flatten(2, a) )
in  reduce2(op +, 0, x)
```

Fusion Statistics	P0	P1	P2	P3	P4	P5
Lines Of Code	283	859	182	23	19	22
`map` ∘ `map`	10	1	8	5	3	
`map` ∘ `replicate`			12	2	2	
`redomap` ∘ `filter`	1					
`redomap` ∘ `map`	5					
`reduce` ∘ `map`	6	12		2	1	1
`reduce` ∘ `replicate`		3				
Interesting		1	1			

Figure 18. Fusion results

which says that a map that reduces each of the input-array elements, followed by a reduce with the same operator (and neutral element) has the semantics of reducing the original array in which the first two dimensions have been flattened. We plan to extend the fusion analysis to incorporate both transformations.

We remark that all transformations presented in this paper, i.e., fusion, ISWIM/IRWIM, REDFLAT, are the result of the rich algebra exposed by the second-order array combinators, and they require a difficult implementation in an imperative context. For example, ISWIM interchanges an inner parallel loop (`map`) outwards across a sequential loop (`scan`). The fact that the inner loop is parallel is not in general sufficient to guarantee the correctness of loop interchange, albeit a parallel loop can always be interchanged inwards.

4.3 Fusion-Analysis Statistics

Since the $\mathcal{L}_0$ language is currently interpreted, we cannot measure directly at this point the effectiveness of our implementation of fusion analysis, which comprises about 1000 lines of Haskell code. We have instrumented the $\mathcal{L}_0$ compiler to keep track of how often and what types of SOAC it successfully fuses. The results are reproduced in Figure 18. We count as "interesting" those fusions in which the number of tuple elements produced by the producer is less than what is used by the consumer. The test programs are:

P0: a real-world pricing kernel for financial derivatives [26],

P1: a real-world market calibration kernel that computes some of the parameters of **P0**,

P2: a real-world kernel for stochastic volatility calibration via Crank-Nicolson finite differences solver [25],

P3: a flat-parallelism, array-based implementation of the shortest-path algorithm (whose shape resembles matrix multiplication),

P4: the flat-parallelism implementation of matrix multiplication shown in Figure 15, i.e., similar to the one of REPA [22],

P5: an implementation of the maximal segment sum problem MSSP.

The results indicate that the `(redo)map` ∘ `map` reductions are the most common, and that there are a significant number of reductions involving `replicate`. The reason for the latter case is that `replicate` is often used to match either the array sizes, as in the case of the flat-parallelism style matrix multiplication, or the result type of a reduce. Our aggressive fusion of `replicate` eliminates these inefficiencies, e.g., it transforms the flat-parallel matrix multiplication to the typical three-level nest of two `map`s and a `redomap`.

There is only one `redomap` ∘ `filter` reduction, appearing in the Sobol random-number generator, albeit an important one: It would increase the parallelism degree by a factor of 32, i.e., the size of the input array, and would also allow efficient computation on GPGPU, i.e., a segmented reduce of a power-of-two size requires only local barriers, rather than multiple execution of a GPU kernel.

Finally, benchmark P0 presents a case where the application of ISWIM would have a significant impact: ISWIM would provide an extra dimension of exploitable parallelism of size 365 on a data set that starves for additional parallelism.

5. Related Work

Loop fusion is an old technique, dating back at least to the seventies [14], with the treatment of loop fusion in a parallel setting being covered in [24]. In imperative languages, the word "fusion" typically does not refer to producer-consumer fusion, but to a complimentary technique, in which two sequential loops that do not depend on each other can fused into a single loop. Single Assignment C [18] incorporated this in a functional language.

The ideas behind a language algebra date back to the very beginnings of functional programming [3], and an algebra that is a subset of ours was presented in [8]. In general, functional language compilers focused on removing intermediate data structures via a structural techique called *deforestation*, which also performs certain kinds of fusion[17].

Data-Parallel Haskell (DPH) [13] makes use of aggressive inlining and rewrite rules to perform fusion, including expressing array operations in terms of streams [16], which have previously been shown to be easily fusable. While DPH obtains good results, rewrite rules are quite limited – they are an inherently local view of the computation, and would be unable to cope with limitations in the presence of in-place array updates, and whether the result of an array operation is used multiple times. The Glasgow Haskell Compiler itself also bases its list fusion on rewrite rules and cross-module inlining [21].

The Repa [22] approach to fusion is based on a delayed representation of arrays, which models an array as a function from index to value. With this representation, fusion happens automatically through function composition, although this can cause duplication of work in many cases. To counteract this, Repa lets the user *force* an array, by which it is converted from the delayed representation to a traditional sequence of values. The pull arrays of Obsidian [15] use a similar mechanism.

Accelerate [23] uses an elaboration of the delayed arrays representation from Repa, and in particular manages to avoid duplicating work. All array operations have a uniform representation as constructors for delayed arrays, on which fusion is performed by tree contraction. Accelerate supports multiple arrays as input to the same array operation (using a `zipWith` construct). Although arrays are usually used at least twice (once for getting the size, once for the data), it does not seem that they can handle the difficult case where the output of an array operation is used as input to two other array operations.

NESL has been extended with a GPU backend [6], for which the authors note that fusion is critical to the performance of the flattened program. Their approach is to use a form of copy-propagation on the intermediary code, and lift the resulting functions to work on entire arrays. Their approach only works for what we would term `map` ∘ `map` fusion, however.

Our uniqueness attributes have some similarities to the "owning pointers" found in the impure language Rust [20], albeit there are deep differences. In Rust, owning pointers are used to manage memory – when an owning pointer goes out of scope, the memory it points to is deallocated – while we use uniqueness attributes to handle side effects. In addition, we allow function calls to consume arrays passed as unique-type parameters, whereas in Rust this causes a deep copy of the object referenced by the owning pointer.

A closer similarity is found in the pure functional language Clean, which contains a sophisticated system of uniqueness typing [5]. Clean employs uniqueness typing to re-use memory in cases where a function receives a unique argument, but also (and perhaps more importantly) to control side effects including arbitrary I/O. As in $\mathcal{L}_0$, alias analysis is used to ensure that uniqueness

properties are not violated. A notable difference is that the Clean language itself does not have any facilities for consuming unique objects, apart from specifying a function parameter as unique, but delegate this to (unsafe) internal functions, that are exposed safely via the type system. Furthermore, a unique return value in Clean may alias some of the parameters to the function, which is forbidden in $\mathcal{L}_0$. We have found that this greatly simplifies analysis, and allows it to be fully intraprocedural.

6. Conclusions and Future Work

Previous work on fusion has taken two main directions: *either* fusion is performed aggressively, and the programmer is provided primitives to inhibit fusion, e.g., forcing array to materialize, *or* fusion is performed via rewriting rules. The latter approach relies tightly on the inliner engine and its applicability is limited to the case when each fused array is consumed by one array combinator.

This paper has presented a program-level, structural-analysis approach to fusion that handles the difficult case in which an array produced by a second-order array combinator (SOAC), such as `map`, is consumed by several other SOACs (if the SOAC producer-consumer dependency graph is reducible.) This essentially allows fusion to operates across `zip`/`unzip`.

Furthermore, we have shown a compositional algebra for fusion that includes array combinators, such as `map`, `reduce`, `filter`, and `redomap`, and other built-in functions that would otherwise hinder fusion applicability, such as `size`, `split`, `transpose`, etc.

Finally, we have discussed two transformation, ISWIM and REDFLAT, that optimize some important uses of `scan` and `reduce`, and that can both enable and be enabled by fusion.

Acknowledgments

This research has been partially supported by the Danish Strategic Research Council, Program Committee for Strategic Growth Technologies, for the research center 'HIPERFIT: Functional High Performance Computing for Financial Information Technology' (http://hiperfit.dk) under contract number 10-092299.

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Pearson Addison Wesley, 2007. ISBN 0-321-49169-6.

[2] R. Allen and K. Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kaufmann, 2002. ISBN 1-55860-286-0.

[3] J. Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.

[4] E. Barendsen and S. Smetsers. Conventional and Uniqueness Typing in Graph Rewrite Systems. In *Found. of Soft. Tech. and Theoretical Comp. Sci. (FSTTCS)*, volume 761 of *LNCS*, pages 41–51, 1993.

[5] E. Barendsen and S. Smetsers. Uniqueness Typing for Functional Languages with Graph Rewriting Semantics. *Mathematical Structures in Computer Science*, 6(6):579–612, 1996.

[6] L. Bergstrom and J. Reppy. Nested Data-Parallelism on the GPU. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*, pages 247–258. ACM, 2012.

[7] R. S. Bird. An Introduction to the Theory of Lists. In *NATO Inst. on Logic of Progr. and Calculi of Discrete Design*, pages 5–42, 1987.

[8] R. S. Bird. Algebraic Identities for Program Calculation. *The Computer Journal*, 32(2):122–126, 1989.

[9] G. Blelloch. Programming Parallel Algorithms. *Communications of the ACM (CACM)*, 39(3):85–97, 1996.

[10] G. E. Blelloch. Scans as Primitive Parallel Operations. *Computers, IEEE Transactions*, 38(11):1526–1538, 1989.

[11] G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a Portable Nested Data-Parallel Language. *Journal of parallel and distributed computing*, 21(1):4–14, 1994.

[12] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Non-Linear Expressions. In *Procs. Int. Conf. on Supercomp. (ICS)*, pages 528–537, 1994.

[13] M. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data Parallel Haskell: A Status Report. In *Int. Work. on Decl. Aspects of Multicore Prog. (DAMP)*, pages 10–18, 2007.

[14] T. E. Cheatham Jr. Programming Language Design Issues. In *Design and Implem. of Prog. Lang.*, pages 399–435. Springer, 1977.

[15] K. Claessen, M. Sheeran, and B. J. Svensson. Expressive Array Constructs in an Embedded GPU Kernel Programming Language. In *Work. on Decl. Aspects of Multicore Prog DAMP*, pages 21–30, 2012.

[16] D. Coutts, D. Stewart, and R. Leshchinskiy. Rewriting Haskell Strings. In *Practical Aspects of Decl. Lang.*, pages 50–64. Springer, 2007.

[17] A. Gill, J. Launchbury, and S. L. Peyton Jones. A Short Cut to Deforestation. In *Procs. of Int. Conf. on Functional Prog. Lang. and Computer Arch.*, pages 223–232. ACM, 1993.

[18] C. Grelck and S.-B. Scholz. SAC - A Functional Array Language for Efficient Multi-Threaded Execution. *International Journal of Parallel Programming*, 34(4):383–427, 2006.

[19] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. Interprocedural Parallelization Analysis in SUIF. *Trans. on Prog. Lang. and Sys. (TOPLAS)*, 27(4):662–731, 2005.

[20] G. Hoare. The Rust Programming Language, June 2013. URL http://www.rust-lang.org/.

[21] S. P. Jones, A. Tolmach, and T. Hoare. Playing by the Rules: Rewriting as a Practical Optimisation Technique in GHC. In *Haskell Workshop*, volume 1, pages 203–233, 2001.

[22] G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, Shape-Polymorphic, Parallel Arrays in Haskell. *ACM Sigplan Notices*, 45(9):261–272, 2010.

[23] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising Purely Functional GPU Programs. In *Procs. of Int. Conf. Funct. Prog. (ICFP)*, 2013.

[24] S. P. Midki and D. A. Padua. Issues in the Compile-Time Optimization of Parallel Programs. In *Procs. of Int. Conf. on Parallel Processing*, volume 2, pages 105–113, 1990.

[25] C. Munk. Introduction to the Numerical Solution of Partial Differential Equations in Finance. 2007.

[26] C. Oancea, C. Andreetta, J. Berthold, A. Frisch, and F. Henglein. Financial Software on GPUs: between Haskell and Fortran. In *Funct. High-Perf. Comp. (FHPC'12)*, 2012.

[27] C. E. Oancea and L. Rauchwerger. Logical Inference Techniques for Loop Parallelization. In *Procs. of Int. Conf. Prog. Lang. Design and Impl. (PLDI)*, pages 509–520, 2012.

[28] C. E. Oancea and L. Rauchwerger. A Hybrid Approach to Proving Memory Reference Monotonicity. In *Int. Lang. Comp. Par. Comp. (LCPC'11)*, volume 7146 of *LNCS*, pages 61–75, 2013.

[29] S. Rus, J. Hoeflinger, and L. Rauchwerger. Hybrid Analysis: Static & Dynamic Memory Reference Analysis. *Int. Journal of Par. Prog*, 31(3):251–283, 2003.