Functional Parallels of Sequential Imperatives (Short Paper)

Tiark Rompf * Kevin J. Brown†

*Purdue University, USA: {firstname}@purdue.edu
†Stanford University, USA: kjbrown@stanford.edu

Abstract

Symbolic parallelism is a fresh look at the decade-old problem of turning sequential, imperative, code into associative reduction kernels, based on the realization that map/reduce is at its core a staging problem: how can programs be separated so that part of the computation can be done before loop-carried dependencies become available? Previous work has investigated dynamic approaches that build symbolic summaries while the actual data is processed. In this short paper, we approach the problem from the static side, and show that with simple syntax- or type-driven transformations, we can readily transform large classes of imperative groupby-aggregate programs into map/reduce parallelism with deterministic overhead.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

Keywords Parallelism, DSLs, staging, map/reduce

1. Introduction

Automatic parallelization of imperative programs is a decade-old and well-studied problem. In the general case, there is only limited success. However, there is mounting evidence that automatic parallelization is tractable for an increasing number of restricted, but practically highly relevant, classes of applications. The Polyhedral Model, for example, works well for applications that are dominated by loop nests over dense arrays [2].

The present work is prompted by a recent paper [36] that uses symbolic execution to effectively parallelize a large class of sequential groupby-aggregate programs, which commonly occur in analytics pipelines that process log files, financial transaction streams, or other time series data. The key idea is to treat stateful variables, which give rise to loop-carried dependencies, as symbolic instead of actual data (e.g. of a symbolic type `SymInt` instead of `Int`). The input data is processed in parallel, and for each chunk a symbolic result is computed, which describes the result of processing the chunk as a formula over the symbolic state variables. These symbolic results are then merged in a parallel tree reduction, replacing the symbolic inputs of chunk $n + 1$, with the symbolic outputs of chunk n, and simplifying. While this approach is completely general, it is of course only viable if the summaries are quick to compute and orders of magnitude smaller than the input data, ideally of small and bounded size. Thus, care is taken to avoid blow-up of formulas: first, by efficient decision procedures that simplify, canonicalize, and merge symbolic results, and second, by restricting the available operations on symbolic values to avoid some that are known to be problematic.

We draw two key lessons from this work [36]: First, map-reduce style processing gives rise to a *staging* problem, where input for the reduce phase becomes available only after the map phase completes. Hence, if future data is required by the map phase, it must be treated symbolically, and the map phase must compute a staged, symbolic result, to be evaluated in the reduce phase. Second, by restricting the types of the symbolic values and their available operations, we can control the shapes of the resulting symbolic formulas.

Armed with these insights, the question bears asking whether one can improve on this model. Traversing a very large dataset and performing symbolic operations, running decision procedures, and simplifying formulas for each data element is certainly more expensive than running a tight loop over the data and performing the native operations directly. And can we really be sure that formulas do not explode in size without airtight static guarantees? So, in concrete terms, we ask: if we keep the stage distinction, and the restricted types and operations, is it really necessary to compute the symbolic summaries based on the actual dynamic data values, or couldn't we obtain comparable results through a static transformation, just by looking at the program text?

In this paper, we show that indeed we can. We extend the approach of [36] to also treat the input data symbolically, but separate from the aggregation variables. We make the following contributions:

- We define a small language for user-defined aggregation kernels that syntactically separates map-phase expressions from reduce-phase expressions. Reduce-phase expressions are (with few exceptions) monoid operations, so that we obtain static guarantees that the programs can be parallelized with deterministic overhead. We do not rely on sophisticated decision procedures and delicate formal reasoning, but phrase our translation as a straightforward syntactic mapping (Section 2).
- Beyond pure monoid operations, we follow [36] in allowing data dependent control flow on symbolic values of bounded domain. Recognizing that we are dealing with a staging problem, we point out that the intuitive solution of [36] and others is an instance of "The Trick" [9, 23], a classic programming pattern in staging and partial evaluation also known as bounded static variation, which computes results from multiple control-flow paths, if the branching decision depends on future-stage values which can be exhaustively enumerated in an earlier stage (Section 3).
- We report on our ongoing work on implementing this model as a DSL on top of Delite and LMS, and provide preliminary performance results. The syntactic restrictions in the formal model translate to data types in the embedded DSL, similar to the symbolic types (e.g. `SymInt`) in the dynamic symbolic execution approach [36] (Section 4).

We survey related work in Section 5.

2. A Core Language for Aggregation Kernels

The epitome of imperative programming are loops with destructive updates, for example:

```
var n = 0
for (d <- data)
    n += d
```

In general, such programs are hard to parallelize, because of the inherent loop-carried dependencies.

Wearing our functional programming glasses, of course, it is easy to see that we could also write this program as:

```
data.reduce(0)(_ + _)
```

Written in this way, the loop-carried dependency has gone away, and `reduce` can be efficiently parallelized: either across multiple CPU cores, using SIMD instructions, and even on GPUs.

The underlying realization is that `(Int,+)` forms a monoid, and therefore the statement `n += d`, taken as syntactic sugar for the assignment `n = n + d`, does not have to be executed from left to right over the input. Instead, since `+` is associative, loop iterations can be executed, and partial results combined, in any order.

2.1 Formal Syntax and Semantics

Let us make things a bit more formal. For starters, we consider the language syntax shown in Figure 1. We let `n` denote accumulator variables, `d` loop variables, and `c` constants. We use `Map[K,V]` to mean the type of efficient map data structures from type `K` to type `V`, and we use `Map(k -> v, ...)` to represent the construction of a map by enumeration of key value pairs, and `s(k -> v)` by updating an existing map `s` with a new binding. We use `A => B` to denote (partial) functions from `A` to `B`. Throughout this paper, we assume syntactically well-formed terms.

We go ahead and assign a sequential semantics to this language in terms of `fold`, following the usual idea that statements are state transformers. The state we compute is a mapping from variable names to integers. The result is shown in Figure 2.

So for our example we have:

```
data.fold(Map(n->0)) { (s,d) => s(n -> s(n) + d) }
```

Now for parallel execution, we would like to rephrase this as:

```
data
  .map(d => Map(n->d))
  .reduce(Map(n->0)) { (l,r) => Map(n -> l(n) + r(n)) }
```

What is the difference between `fold` and `reduce`? Fold has type `(A,E)=>A`, Reduce has type `(A,A)=>A`.

In general, any map/reduce comination can be written as a fold:

```
  data.map(m).reduce(z)(r)
= data.fold(z)((s,d) => r(s,m(d)))
```

So, to go the other way, from fold to reduce, we need a way of breaking apart a fold function `f:(A,E)->A` into `m:E->A` and `r:(A,A)->A` such that

```
f(s,d) = r(s,m(d))
```

and `r` is associative. Crucially, `m` cannot make use of `s` anymore!

In our language this property is enforced syntactically since `n` can never occur in expressions.

We can thus define a parallel semantics in terms of `map` and `reduce` instead of `fold`. The result is shown in Figure 3, and it is easy to verify that the new semantics coincides with the one in Figure 2. Note that sequential composition is just `reduce`.

```
P ::= var n = c; ...; for (d <- data) S   Programs
S ::= n += E | S; S | skip                Statements
E ::= d | c                               Expressions
```

Figure 1. Initial Syntax

```
type Name; type Elem; type State = Map[Name, Int]

I: State = Map(n -> c, ...)

⟦P⟧: State
    ⟦var n = c; ...; for (d <- data) S⟧ =
        data.fold(I) { (s,d) => ⟦S⟧(s,d) }

⟦S⟧: (State,Elem) => State
    ⟦n += E⟧(s,d) = s(n -> s(n) + ⟦E⟧(d))
    ⟦S1; S2⟧(s,d) = ⟦S2⟧(⟦S1⟧(s,d),d)
    ⟦skip⟧(s,d)   = s

⟦E⟧: Elem => Int
    ⟦d⟧(d) = d
    ⟦c⟧(d) = c
```

Figure 2. Sequential Semantics: fold

```
type Name; type Elem; type State = Map[Name, Int]

I: State; Z: State; R: (State,State) => State

I      = Map(n -> c, ...)
Z      = Map(n -> 0, ...)
R(l,r) = Map(n -> l(n) + r(n), ...)

⟦P⟧: State
    ⟦var n = c; ...; for (d <- data) S⟧ =
        I R data.map(d => ⟦S⟧(d)).reduce(Z)(_ R _)

⟦S⟧: Elem => State
    ⟦n += E⟧(d) = Z(n -> ⟦E⟧(d))
    ⟦S1; S2⟧(d) = ⟦S1⟧(d) R ⟦S2⟧(d)
    ⟦skip⟧(d)   = Z

⟦E⟧: Elem => Int
  ⟦d⟧(d) = d
  ⟦c⟧(d) = c
```

Figure 3. Parallel Semantics: map/reduce

How can we enrich the language and keep the property of translating to map/reduce reductions?

First, it is straightforward to add state variables that use other monoids than `(Int,+)`. In the formal semantics, we can partition the set of names according to the monoid, and thus syntactically restrict the available operations.

2.2 Conditionals

We first add conditionals:

```
S ::= ... | if (B) S else S
B ::= E < E | E = E | ...
```

The translation rule is:

```
⟦if (B) S1 else S2⟧(d) =
    if (⟦B⟧(d)) ⟦S1⟧(d) else ⟦S2⟧(d)
```

The rules for boolean expressions ⟦B⟧(d) are standard.

If we have (taking a missing branch as skip):

```
for (d <- data)
    if (b(d)) n += m(d)
```

Then the result will be:

```
data.map(d => if (b(d)) Map(n -> m(d)) else Map(n -> 0))
    .reduce(Z)(_ R _)
```

2.3 Inter-Dependent Variables

So far, we can use variables only in isolation. We cannot, for example, assign the value of one variable to another as part of the loop. Fortunately, there are well-known ways to extend the language in this direction [19, 32]. Let us consider computing Fibonacci numbers as an example, and let us assume for a moment that we evaluate assignments in parallel instead of sequentially, i.e. we do not overwrite a variable before the end of a loop iteration:

```
var n0 = 0
var n1 = 1
for (d <- data) {
  n0 := n1
  n1 += n0 // bulk synchronous semantics
}
```

Now we have accumulator variables on the right hand side of sums and assignments. This is not something that is supported so far.

First, observe what happens when we run the program on symbolic inputs n0,n1. We get the following sequence:

```
n0, n1, n1+n0, 2n1+n0, 3n1+2n0, 5n1+3n0, ...
```

Perhaps surprisingly, the Fibonacci numbers occur as coefficients, independent of the starting values of n0, n1. This means that we can easily parallelize computation. Compute, say, 2 chunks. Set n0, n1 = 0,1 for the first chunk, to obtain n0', n1' on the right of this chunk, and then use those values as n0, n1 for the second chunk.

In fact, since we do not use the input d from data, there is nothing to be gained from parallelism. We can just compute on a single core and use the same result multiple times, which, as one reviewer pointed out, generalizes the repeated-squaring method of computing power. But then again, we could also use the closed form formulation for Fibonacci.

How can we make this systematic? We can generalize from here:

```
for (d <- data) {
  n0 := n1
  n1 += n0
}
```

To computing linear combinations of accumulator variables:

```
for (d <- data) {
  n0 := 0 * n0 + 1 * n1
  n1 := 1 * n0 + 1 * n1
}
```

We can observe that this corresponds to a matrix-vector multiplication:

$$\begin{bmatrix} n0' \\ n1' \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} * \begin{bmatrix} n0 \\ n1 \end{bmatrix}$$

Every loop iteration is a multiplication by a matrix. Thus, if we want to combine multiple loop iterations, we need to build the matrix product.

Fortunately, we know that this is an associative operation, so we can translate it to a reduction!

```
type State  = Map[Name, Int]
type RState = Map[Name, Map[Name, Int] ]

I: State;  F: (State, RState) => State
Z: RState; R: (RState,RState) => RState

I      = Map(n -> c, ...)
F(l,r) = Map(n -> l dot r(n), ...)     // dot product
Z      = Map(n -> Map(n -> 1), ...)    // id matrix
R(l,r) = Map(n -> ...)                 // matrix mult

⟦P⟧: State
    ⟦var n = c; ...; for (d <- data) S⟧ =
        I F data.map(d => ⟦S⟧(d)).reduce(Z)(_ R _)

⟦S⟧: Elem => RState
    ⟦n := E*n2, ...⟧(d) = Z(n -> Map(n2 -> ⟦E⟧(d),..))
    ⟦S1; S2⟧(d)         = ⟦S1⟧(d) R ⟦S2⟧(d)
    ⟦skip⟧(d)           = Z
```

Figure 4. Inter-dependent data flow

The running state for n0 will be the row vector [n0 -> 0, n1 -> 1], and the row vector [n0 -> 1, n1 -> 1] for n1.

While we have previously made the assumption of evaluating assignments in parallel, we actually want to treat assignments sequentially. Therefore, we need to change the Fibonacci implementation as follows:

```
var n0 = 0
var n1 = 1
var n2 = 1
for (d <- data) {
  n0 := n1
  n1 := n2
  n2 += n0 // sequential
}
```

The modified formal semantics is shown in Figure 4. Literals as in n += 1 require using an auxiliary variable.

3. Data-Dependent Control Flow

As an example, we may want to find the first element that matches a predicate and return its index.

```
var found = false
var n = 0
for (d <- data)
  if (!found)
    if (p(d))
      found = true
    else
      n += 1
```

For simplicity, we ignore the possibility of early aborts.

The first option, if we are only interested in the result of found, would be to rewrite the program to use an associative operation:

```
for (d <- data)
  found |= p(d)
```

The second option is to extend the computational model to support mutable state directly. The problem is that the map function needs to access state that is computed only during reduction.

We recognize that this is essentially a staging problem, and in fact it is so common that the partial evaluation community refers to the usual solution plainly as "The Trick" [9, 23]. The technique is also

and more accurately described as *bounded static variation*, and it amounts to pre-computing results from multiple control-flow paths when faced with a control-flow decision that depends on future-stage values.

We go on to allow this now in restricted ways, and use type `Bool => (Bool, RState)` as the type of our monoid. The map function in this example becomes (`count` is the delta added):

```
data
  .map(d => (found =>
    if (found) (found = true, count = 0),
    else       (found = if (p(d)) true else false,
                count = if (p(d)) 0 else 1)))
  .reduce((l,r) =>
     /* reduce both paths independently */ )
```

Reduction is basically function composition. If we stick to functions `Bool => RState'`, then evaluation will be very inefficient, as there is one function composition *per element* in the data. Essentially we're processing all the data in sequence, as the final step!

Fortunately we can do better: we materialize functions into map data structures `Map[Bool, RState']`, through an operation similar to eta-expansion: `f` becomes

```
Map(true -> f(true), false -> f(false)).
```

This ensures that each 'function call', i.e. map lookup, completes in constant time.

The result is a data structure that corresponds to a binary tree. Crucially, the value of `found` after a reduction is uniquely determined by the value of `found` before, so all binary trees will have height one (for one variable).

The semantics with data dependent control flow (on a single boolean variable) are shown in Figure 5. Generalization to multiple variables is straightforward.

4. Outlook and Preliminary Results

Given the approach described, we can almost, but not quite, handle the motivating example given by Raychev et al. and supported in their SYMPLE system [36]:

```
for (e <- events) {
  // look for a search event
  if (!srch_found && is_search(e)) {
    // start counting reviews
    srch_found = true;
    count = 0;
  }
  // count reviews
  if (srch_found && is_review(e))
    count++;
  // on a purchase event
  if (srch_found && is_purchase(e)) {
    // report if count > 10
    if (count > 10)
      ret.push_back(e.item);
    // look for the next search
    srch_found = false;
  }
}; return ret;
```

The two missing bits are symbolic vectors (`ret.push_back`) and inequality comparisons on aggregation variables (`count > 10`).

Symbolic data structures are beyond the scope of this paper, but it is interesting to look at relational comparison operators.

```
type State'  = (b:Bool, ns:State)
type RState' = (b:Bool, ns:RState)
type MRState = Bool => RState'

I': State';  F: (State', MRState') => State'
Z': MRState; R': (MRState,MRState) => MRState

I'      = (false, I)
F'(l,r) = (r(l.b), F(l.ns,r(l.b).ns))
Z'      = st => (st, Z)
R'(l,r) = st => val st' = l(st).b
            (r(st').b, l(st).ns R r(st').ns)

eta f = Map(true -> f(true), false -> f(false))

⟦P⟧: State
    ⟦var b; var n = c; ...; for (d <- data) S0⟧ =
       I F' data.map(d => eta(⟦S0⟧(d)))
                .reduce(Z)(l,r => eta(l R' r))

⟦S⟧: Elem => RState
    ⟦b := B⟧(d)  = st => Z'(⟦B⟧(d))
    ⟦S1; S2⟧(d)  = ⟦S1⟧(d) R' ⟦S2⟧(d)
    ⟦skip⟧(d)    = Z'

    ⟦if (b) S1 else S2⟧(d) =
       st => if (st) ⟦S1⟧(d)(st) else ⟦S2⟧(d)(st)
```

Figure 5. Data-Dependent Control Flow

4.1 Future Work: Relational Operators

Boolean state is always precise: it can only be one out of two alternatives. We can generalize this to other values of small, bounded domains, for example integers less than 10.

Another interesting class of state are variables that can partake in relational comparisons, for example:

```
for (d <- data)
  if (d < min)
    min = d
```

This gives rise to the following transfer function for the `map` phase:

```
data.map(d => (min =>
    if (d < min) d
    else         min))
```

While of course computing the minimum is well known to be an associate operation, it is less clear what restrictions should be put on inequality comparisons in general.

For the minimum operator, a useful input abstraction is to treat `min` as an unknown dynamic value with a known static upper bound `d`. When combining the results during reduction, we have `min <= d1` on the left, and then the next input has to compare `min <= d2`. If we know that `min <= (d1 min d2)` we know that none of the tests will trigger, and `min` will be passed on unmodifed.

We can thus represent the partial summaries as a binary tree data structure, dispatching on the upper bound `d`. Reduction becomes a merge operation on binary trees. But given such a model, what operations are permitted on variables that are compared with inequalities, if we want to insist on being able to bound the depth of such trees statically?

We believe this is an instance of a more general challenge, namely adapting patterns like "The Trick" to scenarios where we have some

partial static knowledge about dynamic data, but cannot enumerate all values exhaustively. In the partial evaluation domain, some related work exists on partially static structures [31], partially static operations [49], and more general partial compution methods such as supercompilation [51].

Sato and Iwasaki [42] tackle a similar problem in their work on automatic parallelization, which is also based on matrix multiplication, and proposes a technique to extract the max operator from user code with conditionals on inequality operators. However, they do not consider bounded static variation in their work.

4.2 Preliminary Performance Results

We have run two preliminary performance experiments with our implementation.

First, we calculate Fibonacci numbers ($\texttt{fib}(2^{30})$). The sequential fold version takes 535ms on a 2012 MacBook Pro (4-core Intel Ivy Bridge at 2.7 GHz). The map/reduce version on a single core takes 548ms, and scales linearly to 4 cores. Thus, the parallel version achieves a speedup of 3.9 over the best sequential version on 4 cores.

Despite having to execute more arithmetic operations (a small sparse matrix multiplication instead of scalar addition), the overall arithmetic intensity is very low, and by generating code with only primitive operations, a compiler can perform many standard optimizations.

Our second benchmark is to search a collection of strings, first for a given string `A`, and when that is found, switch to look for a string `B`, and return how many times string `B` is found. In this scenario, the comparisons are relatively costly: we use strings of length 10, and they are set up to fail only on the last character, every time. The sequential fold version takes 176ms, the sequential reduce version 336ms, since on every iteration it must look for both string `A` and string `B`.

This is a kind of worst-case benchmark. In many cases we expect that some computation can be shared between different branches of computation. For example if we are searching for the same string, a clever compiler such as LMS [39] will perform common subexpression elimination across the branches and perform the string comparison only once. In this case we get the same about 176ms as the sequential version.

5. Related Work

There is a large body of work on automatic parallelization of sequential loops. Some notable works that consider conditionals and other expressive features include [14, 53]. Another line of work has considered the extraction of list homomorphisms [17], which can be uniquely decomposed into map/reduce. Approaches for deriving map/reduce implementations have also been studied in a calculational setting through the notion of filter embedding and semiring fusion [13]. We believe that our work is the first to approach the problem from the angle of defining a language that is parallelizable by construction, and providing a dual sequential and parallel semantics.

Multi-stage programming (MSP, *staging* for short), as established by Taha and Sheard [48] enables programmers to delay evaluation of certain expressions to a generated stage. MetaOCaml [4] implements a classic staging system based on quasi-quotation. Lightweight Modular Staging (LMS) [39] uses types instead of syntax to identify binding times, and generates an intermediate representation instead of target code [38]. LMS draws inspiration from earlier work such as TaskGraph [1], a C++ framework for program generation and optimization. Delite is a compiler framework for embedded DSLs that provides parallelization and heterogeneous code generation on top of LMS [3, 27, 40, 41, 46].

Partial evaluation [23] is an automatic program specialization technique. Some notable systems include DyC [18], for C, JSpec/Tempo [43], the JSC Java Supercompiler [24], and Civet [44]. Bounded static variation ("The Trick") is discussed in the book by Jones, Gomard, and Sestoft [23], and has been related to eta-expansion by Danvy, Malmkjær, and Palsberg [9].

Embedded languages have a long history [26]. Hudak introduced the concept of embedding DSLs as pure libraries [21, 22]. Steele proposed the idea of "growing" a language [45]. The concept of linguistic reuse goes back to Krishnamurthi [25]; Language virtualization to Chafi et al. [7]. The idea of representing an embedded language abstractly as methods (finally tagless) is due to Carette et al. [5] and Hofer et al. [20], going back to much earlier work by Reynolds [37]. Compiling embedded DSLs through dynamically generated ASTs was pioneered by Leijen and Meijer [28] and Elliot et al. [12]. All these works greatly inspired the development of LMS. Haskell is a popular host language for embedded DSLs [16, 47], examples being Accelerate [30], and Nikola [29]. Recent work presents new approaches around quotation and normalization for DSLs [8, 33]. Other performance oriented DSLs include Firepile [34] (Scala), Terra [10, 11] (Lua). Copperhead [6] (Python). Rackets macros [50] provide full control over the syntax and semantics.

Program generators for high-performance code include for example ATLAS [52] (linear algebra), FFTW [15] (discrete fourier transform), and Spiral [35] (general linear transformations).

Acknowledgments

We thank the anonymous reviewers for their helpful suggestions. This research was supported by NSF through awards 1553471 and 1564207.

References

[1] O. Beckmann, A. Houghton, M. R. Mellor, and P. H. J. Kelly. Runtime code generation in C++ as a foundation for domain-specific optimisation. In *Domain-Specific Program Generation*, pages 291–306, 2003.

[2] M. Benabderrahmane, L. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *CC*, volume 6011 of *Lecture Notes in Computer Science*, pages 283–303. Springer, 2010.

[3] K. J. Brown, A. K. Sujeeth, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun. A heterogeneous parallel framework for domain-specific languages. In *PACT*, 2011.

[4] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. GPCE, pages 57–76, 2003.

[5] J. Carette, O. Kiselyov, and C. chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *J. Funct. Program.*, 19(5):509–543, 2009.

[6] B. Catanzaro, M. Garland, and K. Keutzer. Copperhead: compiling an embedded data parallel language. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP, pages 47–56, New York, NY, USA, 2011. ACM.

[7] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. Onward!, 2010.

[8] J. Cheney, S. Lindley, and P. Wadler. A practical theory of language-integrated query. In G. Morrisett and T. Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming,*

ICFP'13, Boston, MA, USA - September 25 - 27, 2013, pages 403–416. ACM, 2013.

[9] O. Danvy, K. Malmkjær, and J. Palsberg. Eta-expansion does the trick. *ACM Trans. Program. Lang. Syst.*, 18(6):730–751, 1996.

[10] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: a multi-stage language for high-performance computing. In H. Boehm and C. Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 105–116. ACM, 2013.

[11] Z. DeVito, D. Ritchie, M. Fisher, A. Aiken, and P. Hanrahan. First-class runtime generation of high-performance types using exotypes. In M. F. P. O'Boyle and K. Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, page 11. ACM, 2014.

[12] C. Elliott, S. Finne, and O. de Moor. Compiling embedded languages. *J. Funct. Program.*, 13(3):455–481, 2003.

[13] K. Emoto, S. Fischer, and Z. Hu. Filter-embedding semiring fusion for programming with mapreduce. *Formal Asp. Comput.*, 24(4-6):623–645, 2012.

[14] A. L. Fisher and A. M. Ghuloum. Parallelizing complex scans and reductions. In *PLDI*, pages 135–146. ACM, 1994.

[15] M. Frigo. A fast fourier transform compiler. In *PLDI*, pages 169–180, 1999.

[16] A. Gill. Domain-specific languages and code synthesis using haskell. *Queue*, 12(4):30:30–30:43, Apr. 2014.

[17] S. Gorlatch. Extracting and implementing list homomorphisms in parallel program development. *Sci. Comput. Program.*, 33(1):1–27, 1999.

[18] B. Grant, M. Mock, M. Philipose, C. Chambers, and S. J. Eggers. Dyc: an expressive annotation-directed dynamic compiler for c. *Theor. Comput. Sci.*, 248(1-2):147–199, 2000.

[19] W. D. Hillis and G. L. S. Jr. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, 1986.

[20] C. Hofer, K. Ostermann, T. Rendel, and A. Moors. Polymorphic embedding of DSLs. In Y. Smaragdakis and J. G. Siek, editors, *GPCE*, pages 137–148. ACM, 2008.

[21] P. Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28, 1996.

[22] P. Hudak. Modular domain specific languages and tools. In *Proceedings of Fifth International Conference on Software Reuse*, pages 134–142, June 1998.

[23] N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial evaluation and automatic program generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[24] A. V. Klimov. A java supercompiler and its application to verification of cache-coherence protocols. In A. Pnueli, I. Virbitskaite, and A. Voronkov, editors, *Ershov Memorial Conference*, volume 5947 of *Lecture Notes in Computer Science*, pages 185–192. Springer, 2009.

[25] S. Krishnamurthi. *Linguistic reuse*. PhD thesis, Computer Science, Rice University, Houston, 2001.

[26] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, 1966.

[27] H. Lee, K. J. Brown, A. K. Sujeeth, H. Chafi, T. Rompf, M. Odersky, and K. Olukotun. Implementing domain-specific languages for heterogeneous parallel computing. *IEEE Micro*, 31(5):42–53, 2011.

[28] D. Leijen and E. Meijer. Domain specific embedded compilers. In *DSL*, pages 109–122, 1999.

[29] G. Mainland and G. Morrisett. Nikola: embedding compiled GPU functions in Haskell. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 67–78, New York, NY, USA, 2010. ACM.

[30] T. L. McDonell, M. M. Chakravarty, G. Keller, and B. Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM.

[31] T. A. Mogensen. Partially static structures in a self-applicable partial evaluator. 1988.

[32] T. Mytkowicz, M. Musuvathi, and W. Schulte. Data-parallel finite-state machines. In *ASPLOS*, pages 529–542. ACM, 2014.

[33] S. Najd, S. Lindley, J. Svenningsson, and P. Wadler. Everything old is new again: Quoted domain specific languages. Technical report, University of Edinburgh, 2015.

[34] N. Nystrom, D. White, and K. Das. Firepile: run-time compilation for GPUs in Scala. In *Proceedings of the 10th ACM international conference on Generative programming and component engineering*, GPCE, pages 107–116, New York, NY, USA, 2011. ACM.

[35] M. Püschel, J. M. F. Moura, B. Singer, J. Xiong, J. Johnson, D. A. Padua, M. M. Veloso, and R. W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[36] V. Raychev, M. Musuvathi, and T. Mytkowicz. Parallelizing user-defined aggregations using symbolic execution. In *SOSP*, pages 153–167. ACM, 2015.

[37] J. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. 1975.

[38] T. Rompf. *Lightweight Modular Staging and Embedded Compilers: Abstraction Without Regret for High-Level High-Performance Programming*. PhD thesis, EPFL, 2012.

[39] T. Rompf and M. Odersky. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6):121–130, 2012.

[40] T. Rompf, A. K. Sujeeth, N. Amin, K. Brown, V. Jovanovic, H. Lee, M. Jonnalagedda, K. Olukotun, and M. Odersky. Optimizing data structures in high-level programs. In *POPL*, 2013.

[41] T. Rompf, A. K. Sujeeth, H. Lee, K. J. Brown, H. Chafi, M. Odersky, and K. Olukotun. Building-blocks for performance oriented DSLs. DSL, 2011.

[42] S. Sato and H. Iwasaki. Automatic parallelization via matrix multiplication. In *PLDI*, pages 470–479. ACM, 2011.

[43] U. P. Schultz, J. L. Lawall, and C. Consel. Automatic program specialization for java. *ACM Trans. Program. Lang. Syst.*, 25(4):452–499, 2003.

[44] A. Shali and W. R. Cook. Hybrid partial evaluation. OOPSLA, pages 375–390, 2011.

[45] G. Steele. Growing a language. *Higher-Order and Symbolic Computation*, 12(3):221–236, 1999.

[46] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun. Composition and reuse with compiled domain-specific languages. In *ECOOP*, 2013.

[47] B. J. Svensson, M. Sheeran, and R. Newton. Design exploration through code-generating DSLs. *Queue*, 12(4):40:40–40:52, Apr. 2014.

[48] W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

[49] P. Thiemann. Partially static operations. In *PEPM*, pages 75–76, 2013.

[50] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 132–141, New York, NY, USA, 2011. ACM.

[51] V. F. Turchin. The concept of a supercompiler. *ACM Trans. Program. Lang. Syst.*, 8(3):292–325, 1986.

[52] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[53] D. N. Xu, S. Khoo, and Z. Hu. Ptype system: A featherweight parallelizability detector. In *APLAS*, volume 3302 of *Lecture Notes in Computer Science*, pages 197–212. Springer, 2004.