Evolutionary Program Transformation

thesis proposal

Cosmin Radoi

1 Introduction

Program transformation is at the core of most software tools and processes. Compilers, including just-in-time compilers for interpreted languages such as JavaScript, transform code from a high-level language to lower-level, machine-executable code. Optimizing compilers, as well as automated and manual refactorings, transform code to improve various aspects of its performance, from time, memory, or energy consumption to softer aspects like maintainability or readability.

1.1 Problem

Despite many years of extensive research, program transformation systems still have trouble generating high performance code in many cases. For example, automatic parallelization works well for specific cases (e.g., stencil code) but fails to scale to more complex programs involving indirect memory references.

For most real-world problems, the optimal program is not reached through one large transformation, but is the result of applying several finer-grained transformations. Failures to generate optimal programs boil down to program transformation systems having difficulty answering the following very basic question:

> *Given a program, a set of program transformations, and a goal, which transformation gets the program closest to the goal?*

As a community, we lack a good answer not because of lack of ingenuity but because the problem is fundamentally hard and it does not have a unique answer. For example, an answer in the context of sequential program optimization may be the exact opposite to an answer in the context of automatic parallelization. Furthermore, a transformation which may not seem like a good idea at a particular step may be required for enabling a more powerful optimization further down the pipeline (see Section 3.1). While more powerful program analysis techniques, better domain specific languages, and further specialization will incrementally alleviate the problem, we propose an orthogonal approach which we believe will provide a leap.

1.2 Proposed Approach

We propose to bypass the problem of choosing the optimal program transformation by simply not making the choice in the first place. At each step, instead of making a set-in-stone choice which reflects some local optimum, we allow the program transformation system to consider many possible transformations. After one step, instead of having one new program, there are several new program variants to consider. Thus, our proposed technique is to search for the globally-optimal program among the program variants reachable via several fine-grained transformation steps.

We believe that our approach is feasible. So far, program transformation research has followed the restricting belief that we should give a single transformation as the answer to the above question. This restriction arose from the need for compilers to be very efficient, i.e., for the compilation time to be low. In other areas, like refactoring, the restriction is rooted in human and technical difficulties, or inconveniences, in dealing with more than one program variant. But the restriction is no longer necessary. It it now technically feasible, and also affordable, to explore multiple transformation paths. On the one hand, cloud computing has made it easy and affordable to access large computational resources—compiling and optimizing code in the cloud may become the norm in the near future. On the other hand, developers have become accustomed to working with multiple variants of their applications using tools such as Git [3] or Mercurial [8].

We call the proposed approach *evolutionary program transformation*, because it enriches program transformation with techniques from the *evolutionary algorithms* [13] field. Drawing inspiration from the biological mechanisms of evolution, evolutionary algorithms build upon the idea that given a population of individuals (which can be anything from constants that need to be optimized to programs), applying environment pressure on them (i.e., removing individuals which score low according to a fitness function) causes natural selection, i.e., the fitness of the population as a whole increases, along with the emergence of particularly fit individuals. New individuals are created by either recombination, when two or more individuals (parents) are used to generate a new individual, or mutation, when a single individual is changed directly to create a new individual. For evolutionary program transformation, the idea is to consider program transformations as mutation operators.

The cross-pollination of program transformation and evolutionary algorithms has been mostly untaped, the notable exceptions being superoptimizers [28, 46], (though they are mostly restricted to loop-free, low-level instruction sequences) and genetic programming [32]. Our approach can be seen as a genetic programming approach with only mutation operators which have strong, specific semantic meaning. This dramatically reduces the space of possible program variants, thus increasing the probability of reaching a good solution.

1.3 Proposed Work

We propose two work goals which will provide a foundation for future evolutionary program transformation research and further clarify its feasibility.

Framework. Our first goal is to build a general evolutionary program transformation framework capable of expressing a wide range of program transformations (compilation, compiler optimizations, parallelization, refactoring, etc.) over a variety of programming languages.

We are implementing our approach as an extension of the $\mathbb{K}$ framework [4, 44]. $\mathbb{K}$ is a rewrite-based framework for specifying executable semantics for programming languages. The framework has been used to give formal semantics for C [24], Java [15], JavaScript [5], PHP [25], Python [9], Verilog [36], and Scheme [35]. Furthermore, $\mathbb{K}$ is an implementation of matching logic [19,43,45], which generalizes both term rewriting and Hoare triples into a unified framework for both operational semantics and program verification.

The missing piece in implementing our approach within $\mathbb{K}$, and thus in achieving our proposed work goal, is a way to specify strategies for searching though the space of possible transformations. Once we have this, we will not only be able to express evolutionary program transformation systems in $\mathbb{K}$, but we will also be able to prove their correctness using $\mathbb{K}$'s verification machinery (based on automatic translation to Coq [14]).

Applications. Our second goal is to evaluate the feasibility of the proposed evolutionary program transformation approach on two real-world problems. We will use the proposed $\mathbb{K}$-based framework to implement the solutions, validating and refining the framework in the process.

In terms of exploring the feasibility of the evolutionary program transformation approach, we implemented a prototype specialized for a particular kind of translation, called MOLD [40], and we showed that it is capable of transforming a representative set of sequential imperative implementations of well-known algorithms (e.g., word count) into efficient functional MapReduce [33] programs.

We will validate and test the new $\mathbb{K}$-based framework by using it to reimplement the Mold translator, and we will use $\mathbb{K}$'s verification capabilities to prove that the translator is correct. Furthermore, in order to test the new $\mathbb{K}$-based framework in a new context, we will build a verified LLVM optimizer. We believe that, at the price of significantly increased compilation time, the evolutionary approach will allow our optimizer to achieve better performance then the official one [6].

2 Background

The proposed approach sits at the intersection of program transformation and evolutionary algorithms.

2.1 Program Transformation

Rewriting strategies. Eelco Visser [52] gives a taxonomy of program transformations, and surveys rewriting strategy approaches. Relevant to our approach are ELAN [16], Maude [18], and Stratego [51], which are the inspiration for our strategy language. Related to the evolutionary approach, Bournez and Kirchner [17] discuss the theoretical aspects of extending ELAN with a probabilistic choice strategy.

2.2 Evolutionary Algorithms

Evolutionary algorithms. Our proposal builds upon techniques developed in the evolutionary algorithms [13] field, and is most akin to genetic programming [32]. The main differentiating aspect from genetic programming is that program transformation mutation operators have strong, specific semantic meaning (see Section 3.1 for an example). Furthermore, there is no crossover operator in the proposed system (though integrating crossover is a possible future extension).

2.3 Existing Approaches

Program synthesis. An extensive body of work concerns the use of program synthesis techniques to generate efficient code. Particularly relevant to our approach is superoptimization, where programs are enumerated and checked against supplied test cases to find a desired code sequence. This may be exhaustive as in the original superoptimizer, goal-oriented [28], or stochastic [46]. In many of these applications, the context is a peephole optimizer that reorders the instructions of a critical inner loop at ISA (instruction set architecture) level. This is also the case for component based program synthesis [27]. In contrast, our approach is not limited to low-level code, but applies to much larger scale transformations, including loops, higher-order functions, and objects.

2.4 Applications

Refactoring. There is also work on refactoring toward parallelism or a more functional form. For example, [23] proposes a refactoring tool to parallelize Java loops, and [26] presents an automated refactoring of Java code to use the Java 8 collection operators. Both approaches transform the original program AST directly and are limited to specific access patterns.

Optimizing compilers. Compiler optimization is an old and well-established research field. Particularly interesting for us are instances where the optimizers have difficulty reaching a good solution. The prevalence of general purpose GPUs has catalyzed the interest in source-to-source transformations from sequential codes to parallel orchestration languages (e.g., OpenMP, OpenCL) or GPU languages (e.g.,, CUDA). In [38] for example, a number of these approaches

are evaluated and a new skeleton-based compiler is described. A common theme in these efforts is the reliance on a programmer to identify and annotate their source code to aid the compiler in generating a suitable and correct parallel implementation. By comparison, in our approach, the system automatically discovers if a loop is suitable for translation into a MapReduce style and applies term rewriting rules to enumerate a number of candidate implementations.

3 Approach

We discovered the need for an evolutionary approach while working on the problem of automatically translating sequential, imperative code to functional, parallel MapReduce. In this section we describe the problem and how evolutionary program transformation proves to be a promising solution.

MapReduce is a programming model which has gained traction both in research and in practice over the last decade. Mainstream MapReduce frameworks [1, 21] provide significant advantages for large-scale distributed parallel computation. In particular, MapReduce frameworks can transparently support fault-tolerance, elastic scaling, and integration with a distributed file system. Additionally, MapReduce has attracted interest as a parallel programming model, independent of difficulties of distributed computation [42]. MapReduce has been shown to be capable to express important parallel algorithms in a number of domains, while still abstracting away low-level details of parallel communication and coordination.

Automatically translating sequential imperative code into a parallel MapReduce framework would be very useful. An effective translation tool could greatly reduce costs when re-targeting legacy sequential code for MapReduce. Furthermore, a translator could simplify the process of targeting MapReduce in new programs: a developer could concentrate on sequential code, letting the translator handle parallel computation.

Translating an imperative loop to the MapReduce model inherits many of the difficulties faced by parallelizing compilers, such as proving loops free of loop-carried dependencies. However, the MapReduce framework differs substantially from a shared-memory parallel loop execution model. Notably, MapReduce implies a *distributed memory* programming model: each mapper and reducer can operate only on data which is "local" to that function. So, an automatic translator must at least partition memory accesses in order to create local mapper and reducer functions which do not rely on shared memory.

Additionally, the communication model in MapReduce is more limited than traditional distributed memory parallel programming with message-passing. Instead, mappers and reducers communicate via a *shuffle* operation, which routes mapper outputs to reducer inputs based on on key fields in the data. These restrictions allow practical MapReduce frameworks to run relatively efficiently at large scale; however, they also introduce constraints on the programmer and challenges for an automatic translator.

3.1 Motivating example

Consider the challenge of automatically generating efficient parallel code for `wordcount`, the problem of counting the number of occurrences of each word in a set of documents. Below is the inner loop (counting the occurrences in a document) of a sequential-style, unoptimized, implementation of the algorithm. The implementation is an intermediate step in transforming the code from a sequential imperative form to MapReduce. Programmers could also write this code directly, especially if they have an imperative programming background. The program iterates, using a `fold` (left) operator, through the given *words* collection accumulating the word occurrence counts into a map. `fold` is a higher-order function taking as arguments a binary function, an initial value, and a collection. At each step, starting with the initial value, the function is applied to the current value and new element of the collection. The result of the function application is the new current value, and it iterates until the entire collection is covered. In our example, starting with an empty map, *updateCount* incrementally takes the current *count* map and a *word*, and returns a new map that is identical to the input map except that the count for the given *word* is incremented.

$$
\begin{aligned}
&\mathit{counts} = \lambda\, \mathit{words}\,. \\
&\quad \texttt{let}\ \mathit{updateCount} = \lambda\, \mathit{count}\ \mathit{word}\,.\, \mathit{count}[\mathit{word}\ \texttt{:=}\ \mathit{count}[\mathit{word}] + 1] \\
&\quad \texttt{in fold}\ \ \mathit{updateCount}\ \ \texttt{EmptyMap}\ \ \mathit{words}
\end{aligned}
$$

The program above exposes no parallelism since the `fold` operation is sequential. There are several approaches we could take to parallelize this code.

Via synchronization. A straightforward approach is to parallelize the `fold` directly by sharing the *count* map between multiple threads, while avoiding races by adding a synchronization mechanism over the *count* map. Synchronization is needed because, due to thread scheduling non-determinism, accesses to the *count* map can happen in any order. This means that at each step the *word* may have any value, independent of the underlying order of the `fold` operator.

Via tiling. Another common way to parallelize such code is by taking advantage of the commutativity of the counting operation in order to tile the `fold` [34], as exemplified below. The input *words* collection is first split into N tiles. Then, the tiles are `map`ped to their word counts using the previous `fold` operator. Finally, the partial word counts, *tiledCounts*, are `fold`ed using the plus ($\oplus$) of a map monoid. $a \oplus b$ is another map with the same keys and the values the sum of their values. If a value is missing in either map, it is replaced by the value type's *zero*, i.e., 0 in our case. The `map` operator can be parallelized or

even distributed.

$$
\begin{array}{l}
\mathit{counts} = \lambda\, \mathit{words}\,. \\
\quad \texttt{let}\ \mathit{updateCount} = \lambda\, \mathit{count}\ \mathit{word}\,.\, \mathit{count}[\mathit{word} := \mathit{count}[\mathit{word}] + 1] \\
\qquad \mathit{tiles} = \texttt{groupInTilesOfSize}\ N\ \mathit{words} \\
\qquad \mathit{tiledCounts} = \texttt{map}\ (\texttt{fold}\ \ \mathit{updateCount}\ \texttt{EmptyMap})\ \mathit{tiles} \\
\quad \texttt{in fold}\oplus\ \texttt{EmptyMap}\ \mathit{tiledCounts}
\end{array}
$$

Via shuffling (MapReduce style). Instead of avoiding the non-linear *word* value, a program can inspect it [20] to reveal parallelism. In our example this is achieved by the `groupBy` operator, which takes a unary function and a collection, clusters the elements of the collection according to the function, and returns a map from function return values to their corresponding clusters. After grouping the collection of words by the identity function, the parts of computation operating on distinct *word* values are independent so they can be executed in parallel. Thus, the `map` operator computing the size of each collection of identical words (via `fold`) is parallelizable and, as there is no sharing, also distributable. As `groupBy` creates collections of identical words, the computation is not yet efficient, but this can be improved by subsequent optimizations.

$$
\begin{array}{l}
\mathit{counts} = \lambda\, \mathit{words}\,. \\
\quad \texttt{let}\ \mathit{grouped} = \texttt{groupBy}\ (\lambda\, x\,.\, x)\ \mathit{words} \\
\quad \texttt{let}\ \mathit{mapF} = \lambda\, \mathit{word}\ \mathit{copies}\,.\, \texttt{fold}\ (\lambda\, \mathit{sum}\ \mathit{word}\,.\, \mathit{sum} + 1\ 0\ \mathit{copies}) \\
\quad \texttt{in map}\ \ \mathit{mapF}\ \ \mathit{grouped}
\end{array}
$$

We see how even for a small snippet of code there are at least three very different parallelization approaches. The best choice between them depends on multiple factors including the runtime platform, data size, surrounding code, other transformations which have already been applied, and, importantly, transformations which could be applied further down the optimization pipeline. Synchronization is generally a good choice in a shared memory environment, tiling offers good performance in most cases, while the group-by approach works well when there is specialized support from the runtime platform (e.g., MapReduce frameworks or FPGAs). But these are just rough guidelines. Actual performance depends on many factors, from CPU speed and cache size to communication cost in a cluster. Furthermore, even for a particular runtime platform, the best performance may be obtained by combining several of the above transformations. For example, the *words* could be tiled to distribute them to multiple nodes, each node using its multiple cores to update a synchronized *counts* tile. Finally, the transformation choice depends on surrounding code and possible future transformations. For example, instead of applying one of these transformations, a loop fusion transformation could be applied to merge the `fold` with another computation, increasing granularity. If the other computation has a different loop dependence structure, it may not be possible to use a `groupBy`

approach anymore. Still, the overall solution may have better performance due to increased granularity.

3.2 Evolutionary Program Transformation

We first attempted to solve the problem above in the traditional way, by trying to make a unique choice between transformations. When the system was not making the right transformation choice, we tried to employ more powerful analysis to generate heuristics that work in most cases. We mostly failed on this path. We could not figure out powerful enough heuristics to obtain consistently good results.

The proposed solution came from dropping the requirement to make a unique choice at each step. We thus allowed our system to explore the space of semantically equivalent programs obtained by applying multiple program transformation rules at each step.

Transformation rules. The basic building blocks of program transformation systems are transformation rules. A rule represents one atomic step in the overall program transformation.

Transformation rules can concisely express complex transformations. For example, the "`fold` to `groupBy`" rule below encodes a very general form of the parallelization-via-*shuffle* transformation explained in the motivating example. We use a vertical notation for rules: the pattern above the bar is rewritten to the term underneath the bar. Capital letters (V, E, and B) are pattern variables and side conditions are on the right. $B[g/r[E]]$ denotes replacing all instances of $r[E]$ in B with g.

$$\frac{\texttt{fold}\ (\lambda\, R\, V\, .\, R[E := B])\ R_0}{\begin{array}{r}(\texttt{map}\, \lambda\, k\, l\, .(\texttt{fold}\ (\lambda\, g\, V\, .\, C)\ R_0[k]\ l)) \\ \circ\ (\texttt{groupBy}\ (\lambda\, V\, .\, E))\end{array}} \quad \begin{array}{l} C = B[g/R[E]] \\ R \notin C \wedge R \notin E \wedge \exists\ V \in E \end{array}$$

The rule matches any `fold` with body an update of a collection at index E. The output code first groups the elements of the collection by the index expression ($\texttt{groupBy}\, \lambda\, V\, .\, E$), then it folds each of the groups using the update expression B from original loop body. `groupBy`'s output is a map from each distinct value of E to the corresponding subset of the input collection. The `map` operation's parameters are k, which binds to the keys of the grouped collection (i.e., evaluations of E), and l which contains a subset of the input collection. The fold starts from the k value of R_0, and folds l using operation C (i.e., the original B with accesses to index E of the old reducer replaced with g, the new parameter corresponding only to the k-index of R).

The side condition requires that R does not appear in either the new expression C or the index expression E. Otherwise, the result of the of computation could depend on `fold`'s evaluation order, so the transformation would not be correct. To avoid grouping by an invariant expression, resulting in a single group, the side condition also requires that E is dependent on V.

Revisiting the original example, the expression below is the program before applying the rule (with a beta reduction applied to better highlight the rule match):

$$\texttt{fold}\ (\lambda\, count\ word\,.\, count[word := count[word] + 1])\ \texttt{EmptyMap}\ words$$

`EmptyMap` matches R_0, *count* matches R, and `count[word]+1` matches B. Both V and E are matched by *word*, but this is just a particularity of this simple example. The rule is general and allows any indirection expression E, as long as the side conditions are satisfied. In our example the expression is rewritten to:

$$((\texttt{map}\ \ \lambda\, k\ l\,.\,\texttt{fold}\ (\lambda\, sum\ word\,.\, sum + 1)\ 0\ l) \circ (\texttt{groupBy}\, id))\ words$$

Transformation strategies. Transformation strategies provide a way of orchestrating complex transformations from basic rules. When the rewrite system defined by the transformation rules is confluent and terminating, there is less need for strategies as the input program is reduced to a normal form, regardless of the order in which rules are applied. Still, many program transformation systems (optimizing compilers, refactorings, etc.) are not confluent, nor terminating. Transformation strategies are thus needed to ensure that the rules are selected in a way which leads to a desirable program, and that the transformation terminates. mite Experimenting with MOLD, we learned that our translator needs to distinguish between what we call *refinement* and *exploration* rewrite rules. *Refinement* rules make definite improvements to the input term, e.g., eliminating a redundant operation such as $\texttt{fold}\ (\lambda\, r\ v\,.\, r)\ r_0$. *Exploration* rules may either improve the original code or bring it to a form that allows other rules to apply, e.g., loop fission is not necessarily an optimization but may allow parallelization of one of the loops.

Exploration rules are treated as transitions between states, i.e., applying a transition rule generates a new state in the system. Refinement rules do not generate new states but are applied exhaustively to the output of an exploration rule. The set of refinement rules form a separate confluent rewrite system. One transition in our translator consists of one application of an exploration rule followed by a complete reduction using the set of refinement rules. In Kiama [47], the Stratego-inspired [51] transformation library we used for our prototype implementation, the orchestration of the above behavior is expressed as the following transformation strategy:

$$\texttt{repeat}(\mathit{explorationRules} <* \texttt{reduce}(\mathit{optimizationRules}))$$

`repeat` instructs the system to apply the argument strategy repeatedly until failure, and return the result of the last successful application. The $<*$ binary operator applies its left argument and, on success, its right argument. The strategy succeeds if both arguments succeed. `reduce` is similar to `repeat`, but the argument is also applied to all sub-terms.

$$
\begin{aligned}
\mathcal{C}(F \circ G) &= \mathcal{C}(F) + \mathcal{C}(G) \\
\mathcal{C}(F(G)) &= \mathcal{C}(F) + \mathcal{C}(G) \\
\mathcal{C}(\langle F, G, ... \rangle) &= \mathcal{C}(F) + \mathcal{C}(G) + ... \\
\mathcal{C}(A[I]) &= C_{get}^{collection} + \mathcal{C}(A) + \mathcal{C}(I) \\
\mathcal{C}(A[K := V]) &= C_{set}^{collection} + \mathcal{C}(A) + \mathcal{C}(K) + \mathcal{C}(V) \\
\mathcal{C}(\texttt{map}\, F) &= C_{init}^{\texttt{map}} + C_{op}^{\texttt{map}} * \mathcal{C}(F) \\
\mathcal{C}(\texttt{fold}\, I\ F) &= \mathcal{C}(I) + C_{init}^{\texttt{fold}} + C_{op}^{\texttt{fold}} * \mathcal{C}(F) \\
\mathcal{C}(\texttt{groupBy}\, F) &= C_{init}^{\texttt{groupBy}} + C_{op}^{\texttt{groupBy}} * \mathcal{C}(F)
\end{aligned}
$$

Figure 1: Fitness function

Evolution strategies. While traditional program transformation rules provide powerful ways to order rule application, they are limited to fixed heuristics. While developing the parallelization rules for our translator, we attempted but failed to find fixed heuristics which would consistently lead the transformation towards a highly optimized program. A transformation which may not seem like a good idea at a particular step may be required for enabling a more powerful subsequent optimization, and the interactions between transformation effects on performance are complex enough that they are hard to predict and encode in a fixed heuristic.

This led us to the need for strategies which, drawing inspiration from evolutionary algorithms [13], guide the transformation by using estimates of code quality, i.e., a fitness function. Mold searches through the state space guided by a fitness function which approximates the runtime performance of a program variant.

Figure 1 shows the fitness function for generating MapReduce programs. The fitness function is a cost estimation function computed recursively over a given term. The cost of function composition/application and tuples is the sum of the cost of their subexpressions. The cost for collection accesses has an extra weight ($C_{get}^{collection}$ and $C_{set}^{collection}$) to encourage access localization. `map` and `fold` operators have an initial cost meant to model the start of a distributed operation ($C_{init}^{\texttt{map}}$, $C_{init}^{\texttt{fold}}$, and $C_{init}^{\texttt{groupBy}}$), and have their operation cost multiplied by a constant ($C_{op}^{\texttt{map}}$, $C_{op}^{\texttt{fold}}$, and $C_{op}^{\texttt{groupBy}}$) representing an approximation for the size of the array. Binary operations are curried, and their function has a constant cost. All other functions have a predefined, constant, cost.

A unique set of constants is used for generating all programs in our evaluation, i.e., the constants are not program-specific. We determined good values by manually running experiments and refining the constants. This process could be automated to find a more precise and possibly platform-specific set of constants. Furthermore, our rough fitness function could be made more precise by applying techniques such as those proposed by Klonatos et al. [30].

The separation of the transformation rules from the evolution strategy makes the approach modular. It is very easy to generate optimized code for a new plat-

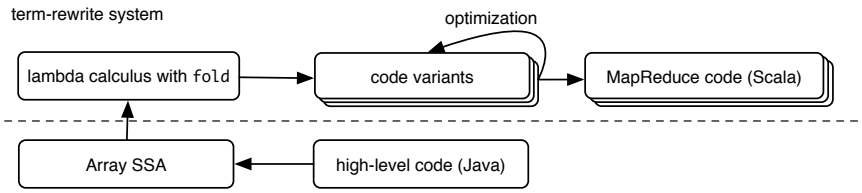

Figure 2: Overview of the Mold translation system.

form by simply adopting a new appropriate fitness function. The transformation rules do not need any adaptation.

3.3 Mold

Figure 2 illustrates the current design of our translator. The input program is first translated into Array SSA form [31] which facilitates the derivation into a lambda-calculus-style functional representation. In contrast with Appel's and Kelsey's work on converting from SSA to functional code [12,29], our translation uses a `fold` operator to maintain the structure of loops, which is essential for later transformations.

The initial "lambda plus `fold`" representation of a program is still far from an effective MapReduce program. While `fold` operations could be implemented using reducers, the lack of mappers hinders parallelism, and the code still operates on shared data structures. To address these problems, we employ a rewriting system to generate a large space of MapReduce programs. The rewrite rules govern where mapper constructs can be introduced in a semantics-preserving manner. Critically, in more complex cases where loop iterations access overlapping locations, we exploit the MapReduce *shuffle* feature to group operations by accessed locations, exposing much more fine-grained parallelism than previous approaches. Given the rewriting rules, our system performs a heuristic search to discover a final output program, using a customizable fitness function to rank programs.

We have implemented our techniques in Mold, a tool that transforms input Java programs into Scala programs that can be executed either on a single computing node via parallel Scala collections, or in a distributed manner using Spark, a popular MapReduce framework [2,53]. Mold leverages the WALA analysis framework [11] to generate Array SSA and implements a custom rewriting engine using the Kiama [47] library. In an experimental evaluation, we tested Mold on a number of input kernels taken from real-world Java and MapReduce benchmarks. In most cases, Mold successfully generated the desired MapReduce code, even for codes with complex indirect array accesses that cannot be handled by previous techniques. To our knowledge, Mold is the first to automatically translate sequential implementations of canonical MapReduce programs like `wordcount` into effective MapReduce programs.

We present some details regarding Mold's implementation by following

through the transformation phases in Figure 2. The translation from Java to Array SSA is an extension of the SSA implementation in WALA [11] to handle arrays and collections. The translation from Array SSA to Lambda IR is implemented in Scala. For the rewrite system we extended Kiama [47] with support for state exploration, name-bindings aware operations (e.g., "subexpression of"), and cost-guided exploration modulo $\alpha\beta$-conversion.

The Scala code is emitted by a pretty printer based on the work of Swierstra and Chitil [48] and on Ramsey's algorithm [41] for minimal parenthesization. The generated code can be executed as traditional Scala code by simple implicit conversions [39]. The same code can be executed either sequentially or using the Scala parallel collections [10]. We also experimented with a backend based on the Spark MapReduce framework [53]. This allows programs generated by MOLD to be executed on Hadoop YARN clusters [50].

3.4 Prototype Evaluation

We experimentally evaluated our prototype transformation tool and the results, while falling short in some regards, are encouraging. We answered the following research questions:

1. **Can we generate *effective* MapReduce code?** By *effective* code we mean code which: 1) should not do redundant computation; 2) should reveal a high level of parallelism; and 3) accesses to large data structures should be localized, to enable execution on a distributed memory MapReduce platform.

2. **Is Mold *efficient* and *general*?** We measure how long it takes for MOLD to find an effective solution. Then, we show that the core rewrite rules are general and match many code scenarios.

To answer these questions, we study the results of using MOLD to translate several sequential implementations of the Phoenix benchmarks [42], a well-established MapReduce benchmark suite which provides both MapReduce and corresponding sequential implementations. The benchmark suite provides C implementations while our system expects Java code as input. We do a manual, straightforward, syntactic, translation of the C sequential implementations to Java.

Furthermore, we manually implemented and tuned each of the benchmarks in Scala to provide a baseline against which we compare the performance of the MOLD-generated code variants. We derived three hand-written implementations: the first uses sequential Scala collections, the second uses parallel Scala collections, and the third uses Spark [53].

To gauge the quality of the code generated by MOLD we pass each program through our tool, we execute the resulting code and hand-written implementations, and we measure and evaluate the results. For each program: 1) we measure MOLD's execution time and log the sequence of applied rules to reach

	generated		hand-written	
program	sequential	parallel	sequential	parallel
WordCount	42.31	17.38	35.58	14.94
Histogram	9.65	7.17	9.98	6.60
LinearRegression	13.77	11.00	13.08	10.60
StringMatch	8.65	3.81	2.58	0.68
MatrixProduct	8.81	2.05	0.41	0.48
PCA	7.02	14.86	3.53	0.77
KMeans	11.90	40.31	0.61	0.89

(a) Scala

	generated		hand-written	
program	1-thread	8-thread	1-thread	8-thread
WordCount	5.99	2.41	5.93	2.42
Histogram	8.84	2.42	8.65	2.59
LinearRegression	1.15	0.62	1.03	0.51
StringMatch	4.78	1.92	4.48	1.57
MatrixProduct	9.02	2.29	5.90	1.58

(b) Spark

Table 1: Execution results.

the optimal solution, 2) we check that the transformations preserved the semantics of the original program by executing both the original program and the generated one on the same input data set and asserting that the results are the same, 3) we manually inspect the code to check whether the operators going over large input data sets have been parallelized, and whether data accesses have been localized, and 4) we execute the generated code with the three execution backends described at the end of Section 3.3, and then we execute the hand-written implementations on the same data sets with the same backends.

When comparing with the hand-written implementations, we execute each version of a benchmark on the same dataset five times within the same JVM instance. We report the average of the measurements after dropping the highest and lowest time values. We run the experiments on a quad-core Intel Core i7 at 2.6 GHz (3720QM) with 16 GB of RAM. Mold's rewrite system state exploration is parallelized.

Can we generate *effective* MapReduce code? We compared the generated code with hand-written implementations for each of the subject programs. As our approach generates more then one solution for each input program, we have inspected not only the top solution, but also the alternate proposed solutions.

Our prototype proposes acceptable solutions for five of the seven subject programs, while falling short for the remaining two. In the first five cases, the top solution is structurally similar to what a programmer would write if targeting MapReduce. In several cases, the alternate proposed solutions are also acceptable. For example, for `wordcount`, the basis for our motivating example (Section 3.1), MOLD generates the MapReduce-style program as the top solution and the tiled solution as an alternative.

For the PCA and KMeans subjects our prototype shows its limitations. While the generated code exposes good parallelism and access localization, it is far from optimal. In PCA, there are three redundant `map`s and one redundant `fold`, out of a total of twelve operators. They are leftover due to limitations of our code motion rules in handling complex tuple structures. Furthermore, in both cases the transformation leaves some additional no-op restructuring of the results at the end of the computation pipeline.

Table 1 compares the execution time of the generated vs. hand-written code. Columns 2 and 3 show the execution time of the generated code using sequential and parallel Scala collections, respectively. Columns 4 and 5 show the hand-written code using sequential and parallel Scala collections, respectively. Columns 6 and 7 show the results of executing the generated code using Spark with either 1 or 8 hardware threads. Columns 8 and 9 show the corresponding results when executing the hand-written Spark implementations.

The generated versions using parallel Scala collections are between 23% and 80% faster than the generated sequential versions, with the exception of PCA and KMeans which exhibit a drastic slowdown when executed in parallel. The generated code for these two benchmarks was found to contain significant redundant computation that severely impacts the performance of the parallel code.

Comparing the executions of the generated and hand-written versions that use parallel Scala collections, we found the former is 4%-16% slower for WordCount, Histogram, and LinearRegression, 4x-6x slower for StringMatch and MatrixProduct, and at least one order of magnitude slower for PCA and KMeans (for the reason explained earlier). The 8-thread Spark-based versions of WordCount, LinearRegression, and StringMatch are faster than the Scala-based ones. We believe Spark gains this advantage by caching intermediate results and by using memory mapping for IO. The generated Spark codes for WordCount and Histogram are slightly faster (1% and 7% respectively) than the hand-written Spark versions. The remaining benchmarks are 22%-45% slower. We lack results for KMeans and PCA because the generated code relies on nested distributed data structures, which are not currently supported in Spark.

Is the approach efficient and general? Table 2 reports the MOLD execution time in column 4. For all but one program, the tool reaches the solution in under 4 minutes, and in some cases, MOLD is fast enough that it could run interactively, with the programmer getting the translated code within tens of seconds. The outlier is KMeans, which is a larger program compared to the others and has separated nested loops and complex control structure. Columns

algorithm	loops	loop nests	translation time (s)	explored variants	transformations (for best variant)
WordCount	2	1	11	944	15
Histogram	1	1	233	34	18
LinearRegression	1	1	28	293	2
StringMatch	1	1	68	554	2
Matrix ×	3	1	40	2779	20
PCA	5	2	66	780	15
KMeans	6	2	340	3982	10

Table 2: Evaluation programs and applied transformations

5 and 6 show the total number of reached distinct program variants, and the number of transformation steps reaching the best variant (according to the fitness function). All transformations included rules exposing parallelism. With the exception of LinearRegression and StringMatch, which reached the optimal solution in two steps, all other programs also require the application of several other complex transformations (code motion, loop fusion and fission, access localization, etc.).

3.5 Limitations

Using MOLD, our prototype implementation, we have shown that the evolutionary approach is effective for parallelizing code. Still, there are several aspects of our implementation that could be significantly improved:

- **generality** –MOLD is limited to the specific problem of translating code from Java to Scala, and cannot be easily extended. We want e a *framework* that easily allows us to develop such MOLD-like systems.
- **ad-hoc transformation rules** – there were several instances where we took advantage of Kiama's seamless implementation with Scala to overcome limitations of Kiama's pattern matching and strategies. While we believe the implementation is correct, deviating from pure rewriting makes it very hard to prove the transformations are correct.
- **flexibility** – the system is a one-way pipeline taking Java bytecode to a WALA intermediate representation, to an intermediate Lambda representation, and finally to Scala. This means we rely on the Java compiler, WALA, and custom transformation code for parts of our pipeline. The mix of technologies makes it difficult to prove that the entire pipeline is correct. Furthermore, we don't have the ability to use our tool as an optimizer of existing Scala code.
- **debugging** – as shown, our prototype did not perform well in two case studies. We attribute this limitations of our code motion rules in handling

complex tuple structures. A more powerful and easy-to-use debugger, such as that available in $\mathbb{K}$, could be very helpful in tracking down the issues.

4 Proposed work

We explain in detail how we plan to leverage the experience we gained while developing the MOLD prototype in order to develop a general evolutionary program transformation framework based on the $\mathbb{K}$ framework [4, 44]. We believe it is possible to overcome MOLD's limitations explained in Section 3.5, and generalize the approach to make it applicable in most program transformation contexts.

Furthermore, to show the generality, we will use the $\mathbb{K}$-based framework to apply the evolutionary program transformation approach in two new contexts: optimizing LLVM code, and improving code readability.

4.1 $\mathbb{K}$-based Evolutionary Transformation Framework

While there are systems implementing projections of the proposed approach for specific problems, we aim to build a general framework capable of expressing a wide range of program transformations (compilation, compiler optimizations, parallelization, refactoring) over various languages, while remaining easy to use and efficient. Such a framework is required to have several characteristics:

1. A programming-language-agnostic way of expressing fine-grained program transformations
2. A way of specifying strategies for searching through the space of possible transformations
3. A way of proving that a transformation is correct (e.g., that it preserves semantics)

We believe that term rewriting provides a powerful foundation for such a framework. Term rewriting systems allow us to naturally and compactly express fine-grained program transformations. All that is needed to define program transformations for a particular but arbitrary programming language is to define the syntax of the that programming language (as well as potentially-needed additional syntax for auxiliary operations as term constructors) and then write transformations as rewrite rules over such terms. There has been extensive work on rewriting strategies, which we will need to extend to include cost-driven strategies inspired from evolutionary algorithms. Finally, rewriting logic [18, 37, 44], has proven to be a solid and practical foundation to giving semantics to programming languages. Such rewrite-based formal semantics can be used to prove that program transformation rules, or at least particular transformation paths, are correct.

The $\mathbb{K}$ framework. We will implement our approach as an extension of $\mathbb{K}$ [4, 44], a rewrite-based framework for specifying executable semantics for programming languages. $\mathbb{K}$ has several characteristics that make it an ideal platform for implementing our approach:

- **maturity and rigor** – $\mathbb{K}$ has been used to give executable formal semantics for C [24], Java [15], JavaScript [5], PHP [25], Python [9], Verilog [36], and Scheme [35]. The C semantics [24] is the most complete and thoroughly tested formal definition of C to date, passing 99.2% of the GCC torture test suite, more then both GCC and Clang (C compiler for LLVM). The JavaScript semantics is the only formal semantics passing the entire ECMAScript 5 conformance test suite (Chrome is the only JavaScript implementation passing all tests, with no other semantics passing more than 90% of them).

- **verification support** – $\mathbb{K}$ is an implementation of matching logic [19,43, 45], which generalizes both term rewriting and Hoare triples into a unified framework for both operational semantics and program verification. Matching logic enables us to use the executable semantics mentioned above in order to prove that specific transformations over those languages are correct (e.g., that they preserve semantics).

Rewriting strategies. We will extend the $\mathbb{K}$ framework with support for transformation strategies. For the strategy language, we will draw inspiration from the excellent work done in developing and incorporating rewrite strategies in rule-based languages like ELAN [16], Stratego [51], and Maude [18].

The $\mathbb{K}$-based framework will allow the strategy specification to have access to the configuration (i.e., the code along with supporting information), and will allow regular transformation rules to change the strategy (similar to ELAN). This is achieved by storing the transformation strategy in a cell of the configuration.

The "execution" of the strategy will be a regular evaluation of the strategy language, just like the execution of any other language defined in $\mathbb{K}$. For example, let us consider the sequencing strategy expression $a <* b$ (explained in the Transformation strategies paragraph of Section 3.2). The strategy expression is stored in the $\langle ... \rangle_{strategy}$ cell of the $\mathbb{K}$ configuration. In the $\mathbb{K}$ method [4], an expression is evaluated by heating (i.e., separating from the rest of the computation) one of its subexpressions, reducing it to a value, and cooling (i.e., replacing) it back into the same location of the original expression.

To allow an equally-powerful evaluation (and execution) of the strategy expression, the $\mathbb{K}$ framework only needs to be extended with one extra behavior: A rule is only tried when its label is at the top of the $\langle ... \rangle_{strategy}$ cell. For our example, the rule labeled a is only tried when the expression a is at the top of the strategy cell, i.e., when strategy cell matches $\langle a \curvearrowright R \rangle_{strategy}$. If the rule is applied, the rule expression is evaluated to $\top$, so the strategy cell becomes $\langle \top \curvearrowright R \rangle_{strategy}$. If the rule fails to apply, the top of the strategy cell is replaced with $\bot$, so the strategy cell becomes $\langle \bot \curvearrowright R \rangle_{strategy}$.

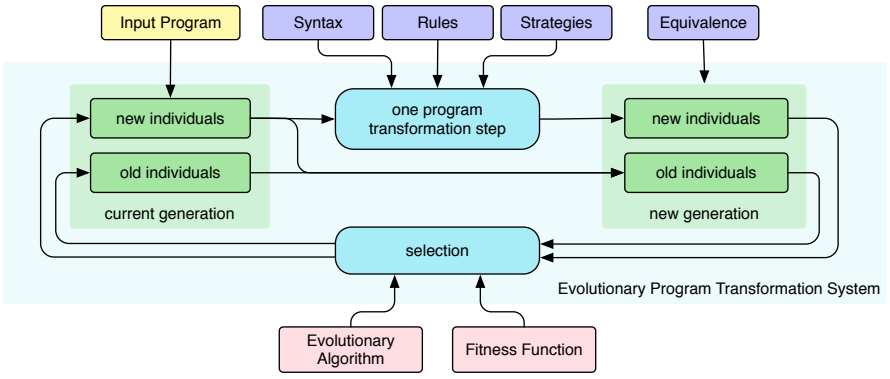

Figure 3: Evolutionary Program Transformation Framework

If the sequencing strategy mentioned above would not be provided by our framework, it could be easily defined by the user:

$$\frac{\langle \bot <* A \curvearrowright R \rangle_{strategy}}{\langle \bot \curvearrowright R \rangle_{strategy}}$$ if the left-hand side fails, the sequencing fails

$$\frac{\langle \top <* B \curvearrowright R \rangle_{strategy}}{\langle B \curvearrowright R \rangle_{strategy}}$$ if the left-hand side succeeds, try the right-hand side

$\bot$ and $\top$ denote a failed and successful strategy, respectively.

The approach is flexible and extensible. The semantics of the strategy language is not hardcoded in the framework, but is defined as any other $\mathbb{K}$ language definition. Users can easily extend the strategy language with new operators, or can even define an entirely new strategy language, without any change to $\mathbb{K}$, the underlying framework.

Evolutionary strategies. The next step is to extend the $\mathbb{K}$ framework with support for evolutionary strategies. Figure 3 shows a high-level overview of the execution of a system built with our proposed framework, highlighting how it blends program transformation with evolutionary algorithms. The program variants are individuals of a population. The population evolves through applications of program transformation rules and strategies, and is guided by an evolutionary algorithm based on a fitness function.

The algorithm is iterative. The initial population consists of just the input program. At each step, for each new individual (i.e. program variant) in the current generation the rewrite system applies all possible transformations. The resulting individuals are considered the new individuals of the new generation. The new and old individuals of the previous generation are added to the new generation as old individuals, without transformation. Next, the new generation is trimmed, selecting only the best-fit individuals according to the given fitness function. After selection, the remaining new and old individuals form the current generation. The algorithm iterates until some termination condition (e.g.,

a fitness improvement ratio compared to the input program, or a time-based cutoff) is reached.

The framework is parametric in the programming language, in the rules and strategies defining the transformation, in the equivalence relation between programs, in the fitness function, and in the evolutionary algorithm. The selection algorithm can be a simple selection of the top-segment of the population, or part of a more complex, multi-objective optimization [22]. This parametric design will allow the framework to be adapted to most program transformation problems without any change to the core code base.

Finally, in order to allow even more flexibility in the configuration of the evolutionary framework, we will expose the selection step as a special nullary operator in the $\mathbb{K}$ strategy language. The selection operator will behave like an identity transformation that is only applied if the current individual is considered fit. If the current individual is not fit, the operator is evaluated to $\bot$ and this line of search is abandoned. Otherwise, if the individual is fit, the operator is evaluated to $\top$ and added to the new generation.

4.2 Applications

Our second major goal is to validate the resulting $\mathbb{K}$-based framework, and evaluate the feasibility of the evolutionary program transformation approach, by applying it to two challenging real-world problems.

Imperative-to-MapReduce translator. We will reimplement the Mold imperative-to-MapReduce translator (presented in Section 3) in order to validate and test our $\mathbb{K}$-based framework. In addition to not being limited to a particular language and to a particular domain, we expect the new implementation to also overcome the existing prototype's limitations (see Section 3.5). Then, using $\mathbb{K}$'s verification capabilities, we will prove that the new, $\mathbb{K}$-based implementation of the Mold translator is correct.

Verified LLVM optimizer. To test our framework in a different context, we will build a verified LLVM optimizer. LLVM [6] is a collection of modular and reusable compiler tools and components. Its core is an intermediate representation, the LLVM IR, and a library of optimizations written on this representation. The imperative-to-MapReduce translator optimizes the code after the translation to a functional form. Building an LLVM optimizer will allow us to experiment with evolutionary program transformation in a new context, that of an imperative programming language.

The syntax and semantics of the LLVM IR have already been partially formulated in K in unpublished work [7]. LLVM provides large suite of optimizations over the LLVM IR, including dead code elimination, constant propagation, function inlining, and strength reduction. There is ongoing work in our group to reimplement a significant subset of the optimizations in $\mathbb{K}$, and verify them.

We will extend this work with more powerful transformations (e.g., loop fusion, tiling, etc.) and integrate it into the evolutionary system. We expect our

system to be competitive with the LLVM optimizer, and we will run experiments comparing it to other approaches, such as polyhedral compilation [49].

4.3 Timeline

summer 2015	(internship)
fall 2015	Mold in $\mathbb{K}$ with improved results for PCA and KMeans
	verify at least the core transformations (fold-to-map and fold-to-groupby)
	start work on the LLVM optimizer
spring 2016	finish the LLVM optimizer prototype
	start work on verifying the LLVM optimizer
summer 2016	(possible internship)
fall 2016	submit optimizer work to PLDI '17 and defend thesis

References

[1] Apache Hadoop. `http://hadoop.apache.org`. Accessed on 1/10/2015.

[2] Apache Spark. `https://spark.apache.org`. Accessed on 1/10/2015.

[3] Git distributed version control system. `http://git-scm.com`. Accessed on 1/10/2015.

[4] K Framework. `http://kframwork.org`. Accessed on 1/10/2015.

[5] KJS: A complete formal semantics of JavaScript. `https://github.com/kframework/javascript-semantics`. Accessed on 1/10/2015.

[6] LLVM. `http://llvm.org`. Accessed on 1/10/2015.

[7] LLVM semantics in k. `https://github.com/davidlazar/llvm-semantics`. Accessed on 1/10/2015.

[8] Mercurial distributed source control management tool. `http://mercurial.selenic.com`.

[9] Pyton semantics in k. `https://code.google.com/p/k-python-semantics/`. Accessed on 1/10/2015.

[10] Scala Parallel Collections. `http://docs.scala-lang.org/overviews/parallel-collections/overview.html`. Accessed on 1/10/2015.

[11] T. J. Watson Libraries for Analysis. `http://wala.sf.net`. Accessed on 1/10/2015.

[12] Andrew W. Appel. SSA is functional programming. *SIGPLAN Not.*, 33(4):17–20, April 1998.

[13] Thomas Back. *Evolutionary algorithms in theory and practice*. Oxford Univ. Press, 1996.

[14] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicaël Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. The coq proof assistant reference manual: Version 6.1. 1997.

[15] Denis Bogdanas and Grigore Roşu. K-java: A complete semantics of java. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 445–456, New York, NY, USA, 2015. ACM.

[16] Peter Borovanskỳ, Claude Kirchner, Hélene Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. An overview of ELAN. *Electronic Notes in Theoretical Computer Science*, 15:55–70, 1998.

[17] Olivier Bournez and Claude Kirchner. Probabilistic rewrite strategies. Applications to ELAN. In *Rewriting techniques and applications*, pages 252–266. Springer, 2002.

[18] M. Clavel, F. Durán, S. Eker, J. Meseguer, P. Lincoln, N. Martí-Oliet, and C. Talcott. *All About Maude, A High-Performance Logical Framework*, volume 4350 of *LNCS*. Springer, 2007.

[19] Andrei Ştefănescu, Ştefan Ciobâcă, Radu Mereuţă, Brandon M. Moore, Traian Florin Şerbănuţă, and Grigore Roşu. All-path reachability logic. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications (RTA-TLCA'14)*, volume 8560 of *LNCS*, pages 425–440. Springer, July 2014.

[20] Raja Das, Mustafa Uysal, Joel Saltz, and Yuan-Shin Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22(3):462–478, September 1994.

[21] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

[22] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*, volume 16. John Wiley & Sons, 2001.

[23] Danny Dig, Mihai Tarce, Cosmin Radoi, Marius Minea, and Ralph Johnson. ReLooper: Refactoring for loop parallelism in Java. In *Proceedings of the 24th ACM SIGPLAN Conference Companion on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 793–794, New York, NY, USA, 2009. ACM.

[24] Chucky Ellison and Grigore Rosu. A formal semantics of C with applications. Technical Report http://hdl.handle.net/2142/17414, University of Illinois, November 2010.

[25] Daniele Filaretti and Sergio Maffeis. An executable formal semantics of php. In *ECOOP. LNCS*, 2014.

[26] Lyle Franklin, Alex Gyori, Jan Lahoda, and Danny Dig. Lambdaficator: From imperative to functional programming through automated refactoring. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1287–1290, Piscataway, NJ, USA, 2013. IEEE Press.

[27] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. Synthesis of loop-free programs. In *Proceedings of the 32Nd ACM*

SIGPLAN Conference on Programming Language Design and Implementation, PLDI '11, pages 62–73, New York, NY, USA, 2011. ACM.

[28] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. ACM.

[29] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN workshop on Intermediate representations*, IR '95, pages 13–22, New York, NY, USA, 1995. ACM.

[30] Yannis Klonatos, Andres Nötzli, Andrej Spielmann, Christoph Koch, and Victor Kuncak. Automatic synthesis of out-of-core algorithms. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 133–144, New York, NY, USA, 2013. ACM.

[31] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM.

[32] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[33] Ralf Lämmel. Google's MapReduce programming model — revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[34] Shih-wei Liao. Parallelizing user-defined and implicit reductions globally on multiprocessors. In *Proceedings of the 11th Asia-Pacific Conference on Advances in Computer Systems Architecture*, ACSAC'06, pages 189–202, Berlin, Heidelberg, 2006. Springer-Verlag.

[35] Patrick Meredith, Mark Hills, and Grigore Roşu. A K definition of scheme. Technical Report Department of Computer Science UIUCDCS-R-2007-2907, University of Illinois at Urbana-Champaign, 2007.

[36] Patrick O'Neil Meredith, Michael Katelman, José Meseguer, and Grigore Roşu. A formal executable semantics of Verilog. In *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*, pages 179–188. IEEE, 2010.

[37] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.

[38] Cedric Nugteren and Henk Corporaal. Introducing Bones: a parallelizing source-to-source compiler based on algorithmic skeletons. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[39] Bruno C.d.S. Oliveira, Adriaan Moors, and Martin Odersky. Type classes as objects and implicits. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 341–360, New York, NY, USA, 2010. ACM.

[40] Cosmin Radoi, Stephen J. Fink, Rodric Rabbah, and Manu Sridharan. Translating imperative code to mapreduce. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '14, pages 909–927, New York, NY, USA, 2014. ACM.

[41] Norman Ramsey. Unparsing expressions with prefix and postfix operators. *Software: Practice and Experience*, 28(12):1327–1356, 1998.

[42] Colby Ranger, Ramanan Raghuraman, Arun Penmetsa, Gary Bradski, and Christos Kozyrakis. Evaluating MapReduce for multi-core and multiprocessor systems. In *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA '07, pages 13–24, Washington, DC, USA, 2007. IEEE Computer Society.

[43] Grigore Roşu, Andrei Ştefănescu, Ştefan Ciobâcă, and Brandon M. Moore. One-path reachability logic. In *LICS'13*. IEEE, 2013.

[44] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[45] Grigore Rosu and Andrei Stefanescu. Checking reachability using matching logic. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 555–574, New York, NY, USA, 2012. ACM.

[46] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 305–316, New York, NY, USA, 2013. ACM.

[47] Anthony M Sloane. Lightweight language processing in Kiama. In *Generative and Transformational Techniques in Software Engineering III*, GTTSE III, pages 408–425. Springer, 2011.

[48] S. doaitse Swierstra and Olaf Chitil. Linear, bounded, functional pretty-printing. *J. Funct. Program.*, 19(1):1–16, January 2009.

[49] Christian Lengauer Tobias Grosser, Armin Groesslinger. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 2012.

[50] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[51] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In *Rewriting techniques and applications*, pages 357–361. Springer, 2001.

[52] Eelco Visser. A survey of rewriting strategies in program transformation systems. *Electronic Notes in Theoretical Computer Science*, 57(0):109 – 143, 2001. {WRS} 2001, 1st International Workshop on Reduction Strategies in Rewriting and Programming.

[53] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.