Graph-based Kernel Recognition for Compiler Guidance

María Rodríguez*,[1]

* ICE, RWTH Aachen University, Templergraben 55, 52056 Aachen, Germany

ABSTRACT

Functional and performance portability as well as maintainability have become dominant issues when developing embedded software, mainly due to the increasing use of heterogenous processor architectures in modern designs. Compilers had to evolve as well to cope with this trend. For example, auto-parallelizing, source-to-source and auto-tuning compilers are active research areas. In this work we present an approach that can supply these kinds of compilers with additional information useful to guide the selection of proper optimization techniques. The approach tries to recognize a known algorithmic kernel for which an optimization technique is proven to be beneficial, even though its implementation differs from the original model of the kernel. To that end, our approach uses a graph-based representation for the kernels to perform the analysis.

KEYWORDS: algorithmic classification; program expression graphs; structural representation

1 Introduction

Naturally, there exists many disparate implementations of the same algorithm that are however functionally equivalent. For example, the code listings below implement the matrix transposition in two slightly different ways.

Listing 1: Reference Implementation

```
int i, j;
for (i = 0; i < w; i++)
  for (j = 0; j < h; j++)
    y[(i*h) + j] = x[i + (w*j)];
```

Listing 2: Another Implementation

```
int i, j;
for (i = 0; i < w; i++)
  for (j = 0; j < h; j++)
    y[i][j] = x[j][i];
```

Is it possible for a compiler to recognize that both implementations have the same functionality? Consider that Listing 1 is supplied within the Texas Instruments DSP library [Ins14] as a functional descriptor of an optimized kernel for matrix transposition. The kernel contains platform-dependent optimizations and intrinsic function calls. On the other hand, Listing 2 is an arbitrary example of a benchmark performing the same. Therefore, if the compiler

[1]E-mail: rodriguez@ice.rwth-aachen.de

recognizes the functional equivalence of both then this fact could guide the developer or the compiler to replace Listing 2 with the corresponding TI's optimized kernel.

This is a simple example use case. Our main idea is to supply compilers with additional information that guides their code transformation decisions, e.g. which ones and in which order to apply them. Our approach has the potential to improve many different kinds of compilers, as for example, auto-parallelizing, source-to-source or auto-tuning compilers.

2 Proposed Approach

The proposed graph-based kernel recognition uses an example code, that we call *model*, to represent a specific functionality. The code piece under recognition is called the *unknown kernel*. The approach looks for functional equivalence among both. Therefore, in the previous example, if Listing 1 serves as the *model* that represents the matrix transposition then Listing 2, which is the *unknown kernel*, is analyzed using Listing 1 to determine if they are functionally equivalent.

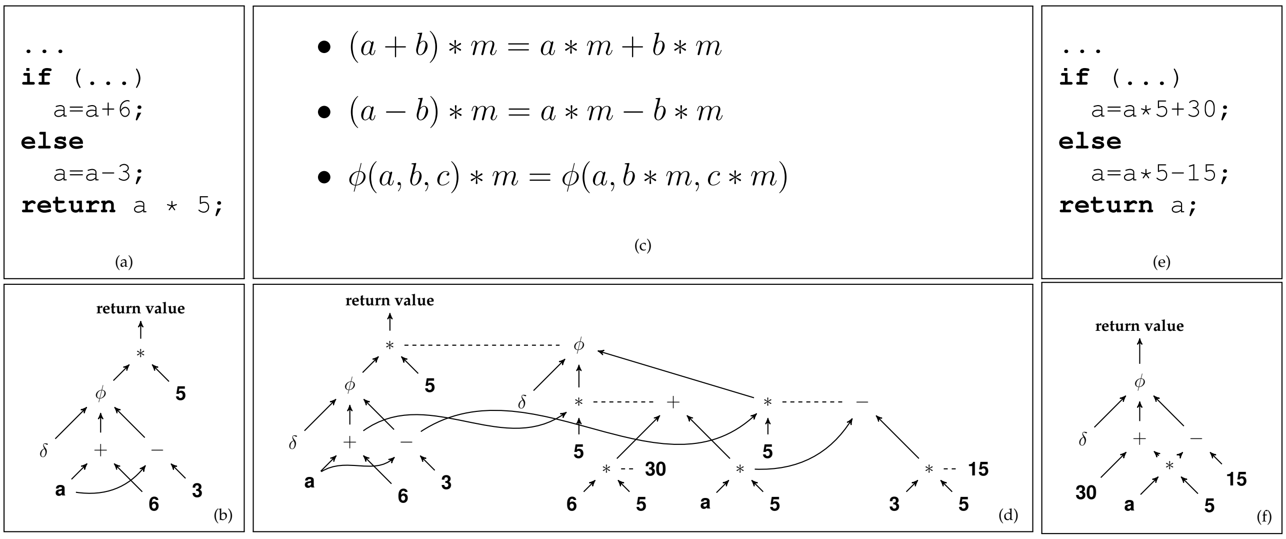
Our first attempt in this direction uses Program Expression Graphs (PEGs) to represent both the *model* and the *unknown kernel*. A PEG is a directed graph where each node represents an operation, incoming edges represent operands, and the outgoing edges represent uses of the result [Gay12]. In [TSTL09], PEGs have been successfully used to represent intraprocedural imperative code with branching and looping constructs.

There are two main reasons to use PEGs. The first one is that PEGs are a complete representation, i.e. no additional structure like a CFG, for example, is needed to represent the original code. Second, PEGs enable equality reasoning. By repeatedly applying a set of simple mathematical rules or axioms one can extend a PEG with information of equivalent expression nodes. The result is a new graph called EPEG. The described process is proposed in [TSTL09], named Equality Saturation. One of the main conclusions of that work is that EPEGs have shown to be an efficient way to represent simultaneously multiple versions of the original program.

The hypothesis of our approach is that after applying Equality Saturation to the PEG of the *unknown kernel*, the result will be an EPEG that contains the model PEG. The recognition process is basically to find within the EPEG of the *unknown kernel* a subgraph that is isomorphic to the model PEG.

In order to illustrate the capabilities of both PEGs and EPEGs, let us consider the example code of Figure 1.a, and its PEG representation in Figure 1.b. At the top of the PEG, the final outgoing edge is the last use of the variable `a`. Most of the nodes are known mathematical operations, except for ϕ. It basically selects depending on the first argument between the second and the third argument, i.e. it represents the merging of the two possible values of `a`. For this example, δ represents the PEG subgraph that computes the branch condition.

By applying the axiom set of Figure 1.c, one can extend the PEG into the EPEG of Figure 1.d. The dashed edges represent the equivalence relations among two expression nodes. It can be thus seen how EPEGs represent simultaneously multiple functionally equivalent code versions. In the example, there are ten ways to select a valid PEG from the EPEG. Figure 1.f shows one of the possible resulting graphs, which is obtained by selecting the proper node in each dashed relation, and Figure 1.e is the corresponding C implementation.



```
...
if (...)
  a=a+6;
else
  a=a-3;
return a * 5;
```

(a)

- $(a+b)*m = a*m + b*m$
- $(a-b)*m = a*m - b*m$
- $\phi(a,b,c)*m = \phi(a, b*m, c*m)$

(c)

```
...
if (...)
  a=a*5+30;
else
  a=a*5-15;
return a;
```

(e)

Figure 1: (a) sample code, (b) the PEG of the sample code, (c) axiom set, (d) EPEG after Equality Saturation, (e) new version of the sample code, and (f) corresponding PEG selection

3 Related Work

What the proposed approach solves can be also considered as a classification problem, where the analyzed code kernels are *objects*, i.e. instances of a certain class, and a *class* is the set of all objects proven to have functional equivalence. Under this view, the graph-based kernel recognition approach basically tries to determine whether or not an *unknown object* belongs to the *class* represented by the *model object*.

As stated in [BR11], to formally define an object is crucial for any classification. This approach uses the *structural way*, in which objects are represented as graphs, instead of the common *statistical way* using a feature vector. The advantage of graphs is that they allow to represent relationships. For example, PEGs describe dependencies and data flow relationships among the computed values, which is a key information for the desired classification. Moreover, unlike feature vectors, graphs are not fixed to a specific size. They can be adapted to the size and complexity of the individual object under consideration.

In the remainder of the section, we summarize related work for algorithm classification. The algorithmic species approach [NCC13], for example, is a fine-grained classification of affine loop nests based on the polyhedral model. They define five fixed classes: element, chunk, neighborhood, shared and fill. These classes describe a specific access pattern for a single array access in a statement. The key idea is the use of these access patterns as building blocks to cover a complete loop nest, which enables the creation of an unlimited amount of different species using only a limited set of access patterns. Therefore, species are classes of algorithms, capturing information about the structure of parallelism and the amount of data re-use in nested for-loops. Using this information programmers can reason about the code or compilers can take decisions.

Our work shares similar goals to the algorithmic species approach, both try to extract algorithm details by means of classifying the analyzed source code. In [NCC13], other classification approaches like dwarfs [ABD$^+$09], the Galois classification system [PNK$^+$11], and algorithmic skeletons [Col91] have been evaluated with respect to five requirements that in their perspective an algorithm classification must meet. The algorithm classes are required to be: (1) automatically extracted, (2) intuitive, (3) formally defined (4) complete and (5) fine-grained. The algorithmic species approach meets all of them. In terms of those requirements, the conclusions for our approach are:

(1) The automatic extraction of classes is possible, since all involved steps are deterministic and formal. There are tools available to automatically obtain PEGs, EPEGs and also to find subgraph isomorphisms.

(2)-(5) The classes in our approach can be as intuitive or complex as the user wants, since the starting point is any C code that describes a functionality. Therefore, the flexibility of our approach enables the exploration of different domains and different levels of granularity.

(3) The PEG is actually a mathematical structure and the relation of equivalence is what defines a class. Therefore, it is also formally sound.

(4) The algorithmic species approach is only complete for the domain of affine nests. Our classification is also complete, although to a larger extent, since it is able to classify any code for which a PEG can be generated. Therefore, our approach can work on algorithms using data structures such as sparse matrices, graphs and trees, for example.

4 Current Work

The first milestone is to evaluate the practicability of the proposed approach, in this direction some challenges have been identified. One consists to define the subgraph isomorphism problem in the presence of the equivalence and data flow edges, as current algorithms work only with one edge type. Additionally, the selection of the axiom set is crucial, as it will define the coverage and efficiency of our approach. Thus, one open question is, which is the minimum axiom set that enables to efficiently recognize functionally equivalent algorithms. Furthermore, we are designing a suitable benchmark for the evaluation of our approach and the identification of potential additional challenges.

References

[ABD$^+$09] Krste Asanovic, Rastislav Bodik, James Demmel, Tony Keaveny, Kurt Keutzer, John Kubiatowicz, Nelson Morgan, David Patterson, Koushik Sen, John Wawrzynek, David Wessel, and Katherine Yelick. A view of the parallel computing landscape. *Commun. ACM*, 52(10):56–67, October 2009.

[BR11] Horst Bunke and Kaspar Riesen. Recent advances in graph-based pattern recognition with applications in document analysis. *Pattern Recogn.*, 44(5):1057–1067, May 2011.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA, USA, 1991.

[Gay12] DavidM. Gay. Using expression graphs in optimization algorithms. In Jon Lee and Sven Leyffer, editors, *Mixed Integer Nonlinear Programming*, volume 154 of *The IMA Volumes in Mathematics and its Applications*, pages 247–262. Springer New York, 2012.

[Ins14] Texas Instruments. Tms320c6000 dsp library (dsplib), 2014.

[NCC13] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4):40:1–40:25, January 2013.

[PNK$^+$11] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. The tao of parallelism in algorithms. *SIGPLAN Not.*, 46(6):12–25, June 2011.

[TSTL09] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. *SIGPLAN Not.*, 44(1):264–276, January 2009.