# Towards a Versatile Pointer Analysis Framework*

R. Castillo, A. Tineo, F. Corbera, A. Navarro, R. Asenjo, and E.L. Zapata

Dpt. of Computer Architecture, University of Málaga,
Complejo Tecnologico, Campus de Teatinos, E-29071. Málaga, Spain
{rosa, tineo, corbera, angeles, asenjo, ezapata}@ac.uma.es

**Abstract.** Current pointer analysis techniques fail to find parallelism in heap accesses. However, some of them are still capable of obtaining valuable information about the way dynamic memory is used in pointer-based programs. It would be desirable to have a unified framework with a broadened perspective that can take the best out of available techniques and compensate for their weaknesses. We present an early view of such a framework, featuring a graph-based shape analysis technique. We describe some early experiments that obtain detailed information about how dynamic memory arranges in the heap. Furthermore, we document how def-use information can be used to greatly optimize shape analysis.

## 1 Introduction

Pointer analysis is a field of study that has drawn a great deal of attention over the past few years. The problem of calculating pointer-induced aliases must be solved so that compilers can safely disambiguate memory references. Static knowledge of pointer-aliasing is key to perform optimizations related to parallelism and locality. While stack-pointer and array aliases allow for successful techniques to be applied, heap-directed pointers render such techniques ineffective. Therefore, new approaches must be taken.

We present in this work a pointer analysis framework that can accommodate several pointer analysis techniques, both existent and new. It is designed as an extensible framework based in Java. High-level program transformations are favored with the use of a near-source IR obtained with *Cetus* [1], a parsing tool aimed towards source-to-source translations. A key part of the framework is a newly designed graph-based shape analysis algorithm [2], that can obtain very detailed information about the arrangement of recursive data structures in the heap. Section 2 introduces our shape analysis technique in the context of the overall framework.

To better understand the shape analyzer capabilities, we have conducted some preliminary tests with typical heap-directed structures. These tests prove that memory configurations are accurately captured. We even discovered that the

* This work was supported in part by the Ministry of Education of Spain under contract TIC2003-06623.

W.E. Nagel et al. (Eds.): Euro-Par 2006, LNCS 4128, pp. 323–333, 2006.
© Springer-Verlag Berlin Heidelberg 2006

analysis times can be greatly reduced by driving the analysis with def-use information. Section 3 documents our experiments with the shape analyzer.

On its own, the shape analysis is a great tool for programmer support, as it can be used by developers to check how dynamic structures are really used in their programs. Better still, higher-level client analysis modules can be built over the shape analyzer. In particular, we focus in dependence analysis in the context of loops that traverse dynamic recursive data structures. Such dependence analysis is needed for automatic parallelization of pointer-based programs and for locality exploitation, which are the final goals of our research. As a prerequisite for the dependence test, we need to automatically detect induction pointers. Section 4 covers the dependence test as a client analysis and how we tackle the automatic detection of induction pointers.

Finally, Section 5 comments some related work and Section 6 concludes with the main contributions and ideas for future work.

## 2 Shape Analysis Within the General Framework

Fig. 1 gives an overview of the general layout for our pointer analysis framework. First, we take an input program and parse it with the Cetus tool. Cetus is a compiler infrastructure specially aimed towards the development of compilation passes of high-level nature. It is written in Java and its source code is publicly available under a non-restrictive license. Cetus can parse C, C++ and soon Java, to a unique *intermediate representation* or IR, where transformations can be performed. Cetus IR is regarded to be close the source code, which is suitable for transformations related to pointer analysis.

Within Cetus, we can design compilation passes that are required by the pointer analysis techniques that follow. Such passes would perform preconditioning transformations, like expression simplification, statement reordering, etc., or would extract information, like data types, CFG, etc., as needed by the subsequent analysis. The results of the analysis can then be used to perform optimizations related to parallelism or locality, modifying the original program to obtain an optimized version.

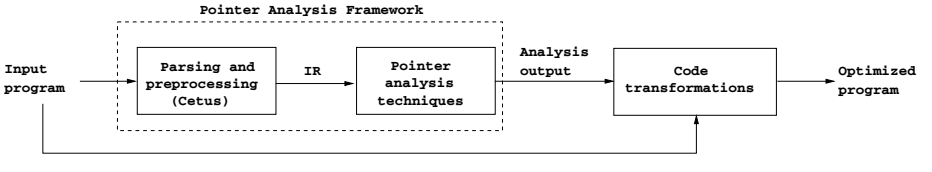


**Fig. 1.** General layout for the pointer analysis framework

Currently, we are focusing on the preprocessing and analysis phases. They conform the pointer analysis framework. Later, we can concentrate on using the results of the different pointer analysis techniques implemented to generate threaded versions of the programs.

The shape analyzer tool [2] is a cornerstone of our pointer analysis framework. Due to space limitations, only the main features and design principles will be described. It provides detailed information about the arrangement of memory locations in the heap for pointer-based programs. That information can be used for several purposes like: (i) data dependence analysis, by determining if two accesses may reach the same memory location; (ii) locality exploitation, by capturing the way memory locations are traversed to determine when are they likely to be contiguous in memory; and (iii) programmer support, to help detecting incorrect pointer usage or documenting complex data structures.

Our shape analyzer works as an iterative data-flow algorithm. It is flow-sensitive, context-sensitive and field-sensitive, although it lacks proper interprocedural support at the current state (we plan to add complete interprocedural support in the near future) and thus functions bodies must be inlined. The algorithm works by performing *abstract interpretation* over the pointer statements in the program until a *fixed-point* is reached. As result of the analysis, shape graphs are generated. Such graphs capture memory configurations arising in the heap in a conservative way. Fig. 2 shows an outline of the algorithm operation in the presence of (a) loops and (b) pointer statements, such as `ptr = ptr2` or `ptr = ptr2->sel`.

**(a) Loop statement class**

```
fun run(ShapeGraph sg)
    ShapeGraph oldSummary = EMPTYGRAPH;
    ShapeGraph newSummary = sg.copy();
    while(newSummary != oldSummary)
        Statement nextStmt = StatementList.next();
        while(nextStmt != NULL)
            sg = nextStmt.run(sg);
            nextStmt = StatementList.next();
      oldSummary = newSummary;
      newSummary.join(sg);
    return newSummary;
    //Return overall effect of loop
```

**(b) Pointer statement class**

```
fun run(ShapeGraph sg)
    ShapeGraphSet sgs = sg.splitBySel();
    //Breaks into possible graphs
    foreach(sg in sgs)
        sg.materializeNode();
        //Focus over currently accessed node
        sg.abstractSemantics();
        //Apply semantics of pointer statement
        sg.normalize();
        //Summarize compatible nodes
    foreach(sg in sgs)
        sgOut.join(sg);
    return sgOut;
```

**Fig. 2.** Outline of shape analysis algorithm regarding loops and pointer statements

Shape graphs are formed by nodes, links and CLSs (Coexistent Links Sets), which codify possibilities of connectivity between memory locations in the program. Graphs change according to the *abstract semantics* of the pointer statements present in the program. Fig. 3 sketches how graph change when analysing the first five statements in the creation of a singly-linked list. Dynamically allocated memory pieces are represented by nodes, and joined together with links. The last graph is also accompanied by its CLSs description, showing the combination of links that are possible for each node.

At compile time, the size and connectivity of recursive data structures is usually unknown. However, our representation of such structures must be finite, i.e., we must provide mechanisms to capture all possible memory configurations arising in the program in a finite number of bounded-size graphs. Graphs are

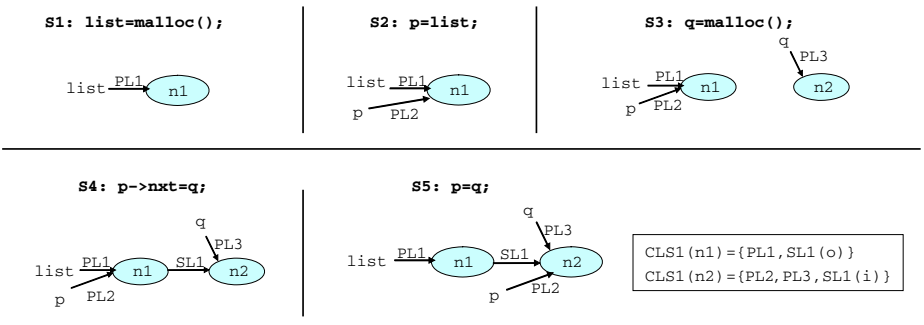


**Fig. 3.** Graphs are modified according to the abstract semantics of each statement

assured to be bounded by the *summarization* process: whenever nodes are regarded as *similar enough*, they are merged in to a so-called summary nodes. Similarity is determined by pointer alias relationships and adjustable *properties*. In fact, properties are a key instrument to fine-tune summarization decisions and therefore control how precisely graphs capture the features of the memory configuration.

Summarizing implies loosing information in favor of a bounded representation. We provide as well a dual operation to focus over previously summarized nodes: *materialization*. This operation can regain precision where pointer accesses are occurring because it performs *strong update* [3] [4], discarding unnecessary links in most situations. However, highly connected and summarized graphs can make impossible for the materialization operation to recover exactly the intended links, leaving some conservative ones.

Our analysis computes all possible memory configurations for every statement in the program. At any point during the analysis, there can be several graphs per statement to reflect all possible memory configurations that can reach the statement from different control flow paths. Different graphs represent mutually exclusive pointer arrangements over memory. Since the number of stack-declared pointer variables is fixed and known at compile time, the number of graphs per statement is limited by the different and mutually exclusive combinations of pointer over nodes and their properties.

The shape analyzer tool has been written in Java, taking in all new features of the latest Java 1.5 release. A big effort has been spent in making this tool as robust as it can be, so the object-oriented approach seemed a natural choice. Developing in such a manner facilitates writing extensions and performing maintenance tasks. Besides, a Java design makes it easier to blend with the extended version of Cetus that serves as front-end for the pointer analysis framework.

Fig. 4 is a simplified view of how elements interact within the pointer analysis framework: first, the input program is parsed by Cetus, this way we achieve an IR where we can easily operate; second, our specific preprocessing pass is run over Cetus IR to translate the program to the format required by the shape analyzer; third, the shape analyzer outputs the graphs for the program, which

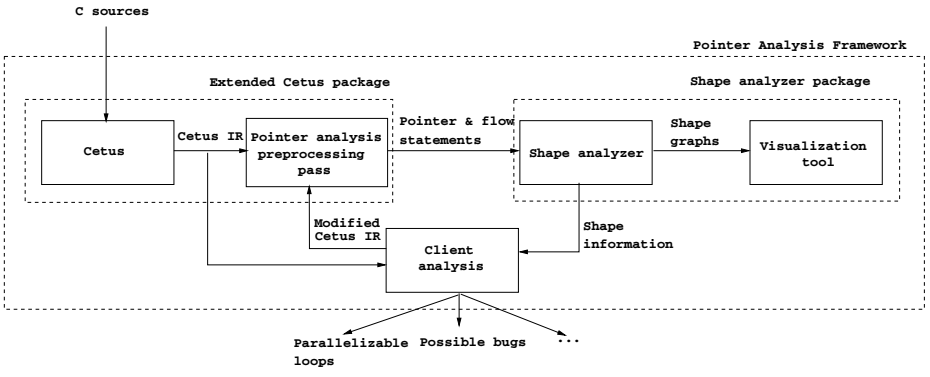


**Fig. 4.** Different modules working together within the pointer analysis framework

can be viewed in the companion visualization tool. Finally, client analysis techniques can be added to produce output results based on shape information, like parallelizable loops, possible bugs, etc. These techniques can drive the analysis to make it more effective as we will see later.

## 3 Experimental Results

We present now some early experimental results regarding the shape analyzer. For these tests we have considered six programs. The first four are typical kernels of applications that deal with recursive data structures. For the last two tests, we consider the product of a sparse matrix by a sparse vector, first based on singly-linked lists, then based on doubly-linked lists. Sparse structures are usually built with pointers to avoid wasting storage capacity with many empty values. Table 1 describes the structures tested and displays some metrics for the analysis performed.

The first column identifies each test, while the second column holds the number of analyzed statements. All available pointer and flow statements are considered. The tests that consider linked lists (singly-linked and doubly-linked) first create the lists, then traverse them. The tests working with trees (n-ary and binary) perform structure traversing during the trees creation, as each new element is added as a leave starting from the root. The sparse matrix is created as a header list (rows), whose elements point to other lists (columns), while the sparse vectors are created as lists. In the fifth test, the structures are based in simply-linked lists (s), while on the sixth test, they are based on doubly-linked lists (d). Regarding the product algorithm, first the input matrix and vector are created, then the output vector is built as the matrix and input vector are traversed. The output for each test is a graph that captures the structures created and traversed. The complete codes and resulting graphs are available through our website[1].

[1] http://www.ac.uma.es/∼asenjo/research/codes.html

**Table 1.** Structures tested in the shape analyzer, number of analyzed statements, time spent on the analysis, total number of generated graphs, and nodes, links and CLSs per graph, in average (and maximum) values

| Data structure | # stmts | Time | # graphs | Nodes, links & CLSs per graph |
|---|---|---|---|---|
| Singly-linked list | 17 | 0.47 sec | 62 | 2.51 (4) / 3.64 (7) / 4.75 (13) |
| Doubly-linked list | 19 | 0.52 sec | 74 | 2.59 (4) / 6.90 (13) / 4.55 (13) |
| N-ary tree | 17 | 0.62 sec | 372 | 2.61 (4) / 6.39 (12) / 9.38 (22) |
| Binary tree | 25 | 2.02 sec | 435 | 2.73 (4) / 10.58 (20) / 23.84 (65) |
| Matrix-vector(s) | 83 | 1.14 min | 2477 | 7.56 (12) / 26.10 (40) / 29.34 (50) |
| Matrix-vector(d) | 97 | 1.55 min | 2931 | 7.60 (12) / 30.95 (48) / 30.37 (50) |

The third column shows times for the tests. Only the time for the actual shape analysis is shown (no parsing or preprocessing), as measured in a Pentium IV 2.4 GHz with 1 GB RAM. We think that times are very reasonable for such a detailed analysis. Within the first four examples of synthetic codes, the highest time is that of the binary tree analysis, probably due to its more complex CFG. It should be noted that more possible flow paths make the analysis more costly, as it has to consider all possibilities conservatively. On the other hand, the first three examples run in less than a second. The matrix by vector product takes longer, clocking at more than 1 minute, which is only reasonable considering there are quite some more statements to analyze than in previous tests.

The fourth column indicates the total number of graphs generated for each test. This metric gives an idea about the internal cost of analyzing different structures and traversals. The numbers range from a few dozens to a few thousands. Next columns show the total number of nodes, links and CLSs per graph, as average values with the maximum in brackets. The number of nodes per graph is essentially constant in the first four tests, as it depends mostly on the number of simultaneously live pointers, which is usually one for the structure handle and two for navigating it. The matrix by vector test has three times more nodes because there are three different structures, instead of one. The number of links depends on the amount of different links that each element has. Typically each element in a recursive data structure does not have more than two links.

Finally, CLSs are the elements where most of the complexity reside: they describe how nodes and links can combine to create all possible memory configurations arising in the program. The highest maximum is for the binary tree among all tests, but the maximum average is attained in the matrix by vector program based on doubly-linked lists.

To sum up, we can say that the shape analyzer can effectively analyze common data structures for pointer-based codes. Generated graphs accurately capture heap structures. Furthermore, we think that such graphs can be obtained in manageable times, specially for such a complex technique. Let us not forget that we are performing fixed-point abstract interpretation of pointer and flow statements to create and modify very detailed graphs. Despite this encouraging

results, it is clear that this is a costly technique which is not likely to succeed if used for whole program analysis. Instead it would be better used within a client analysis module that would focus on *local analysis*.

In this regard, we discovered that def-use information can be used to identify the statements directly involved in the creation of the recursive data structures that are traversed in the segment of code under analysis. A def-use chain establishes a relationship between the definition point where a value is created and points where it is used. With that information we can automatically determine what are the statements that actually define the shape of dynamic memory and discard all other statements. With this approach we avoid to analyze irrelevant statements that could slow down the shape analysis.

We have tried this approach on the matrix by vector examples. Let us revisit them now, having *pruned* all traversal statements that are not involved in the output vector creation. The new values for the tests are shown in table 2, where the original values for the unprocessed versions are also displayed for reference.

**Table 2.** The matrix by vector product analyzed in original (o) and pruned (p) forms, based in singly-linked (s) or doubly-linked (d) lists

| Data structure | # stmts | Time | # graphs | Nodes, links & CLSs per graph |
|---|---|---|---|---|
| Matrix-vector(o,s) | 83 | 1.14 min | 2477 | 7.56 (12) / 26.10 (40) / 29.34 (50) |
| Matrix-vector(p,s) | 66 | 7.52 sec | 772 | 5.69 (10) / 19.28 (36) / 19.91 (48) |
| Matrix-vector(o,d) | 97 | 1.55 min | 2931 | 7.60 (12) / 30.95 (48) / 30.37 (50) |
| Matrix-vector(p,d) | 77 | 9.22 sec | 823 | 5.45 (10) / 21.29 (42) / 19.68 (48) |

The results prove that def-use driven shape analysis works best, as the analysis time has been reduced dramatically. Pruned tests produce the same output graphs than their original counterparts, thus capturing memory configuration without any loss in precision. This example motivates us to tightly integrate shape analysis within client analysis that focus on the statements of interest.

When trying to include def-use chains generation within the framework, we realized that their computation is easier in the SSA form of the program. This led us to implement SSA support as a Cetus extension. The cost of providing SSA support within the framework is not only justified by its use to drive shape analysis. It is also a required module for other pointer analysis and optimizations techniques. In our approach to SSA, we obtain the *dominator tree* in the first place. Then a slightly modified version of Cytron's algorithm [5] is used for constructing the SSA form. We modified the algorithm to remove unnecessary $\phi$-functions that could hinder client analysis performance. We also made it more efficient by renaming each $\phi$-function just once, instead of twice.

## 4 Dependence Test as a Client Analysis

The shape analysis algorithm is a basic element in our pointer analysis framework. However, we are aware that it is not sufficient as a stand-alone analysis

technique. In order to take full advantage of its power it must be coupled with a higher-level client analysis that can determine regions to be analyzed for a given purpose. One of such client analysis is data dependence analysis for loops that traverse dynamic recursive data structures. Ultimately, this kind of analysis can determine what loops can be safely parallelized in an automated basis.

Let us focus now on a common situation. Usually, data structures are created at the initialization phase of programs and later, they are traversed to perform certain calculations. Often, most of the execution time of the program occurs at such traversals, where the structure change no more, but their values do. In such scenario, a client analysis could identify the statements that create the structure, call the shape analyzer over those statements to obtain shape information, and then use that information to look for data dependencies in the loops of interest.

In fact, the dependence detection can also be performed with the help of the shape analyzer, by marking or *touching* traversed nodes with accessing information. More precisely, nodes in the graph can be marked as having been read or written. This is achieved by using the *touch property* in the context of a loop-carried dependence test, similarly to [6]. As a prerequisite of such dependence analysis, induction pointers must be identified for loops that traverse recursive data structures.

*Induction pointers*, also called *navigator pointers*, are used in loops to traverse recursive structures, establishing their traversal pattern. That pattern, along with the shape of the structure, allows to detect dependencies between accesses. Of course, induction pointers already introduce an inherent dependence between different iterations of a loop, something known as the *pointer-chasing problem*. However, there are techniques to overcome it, provided that no other dependencies exist. In the next example, `p` is the *induction pointer*.

```
while (p!= NULL){
    p -> x = p -> y * 5;
    p = p -> next;
}
```

Being able to automatically detect induction pointers is a must for our compiler analysis framework, because they are needed to identify loops that traverse recursive data structures, and thus are candidate for parallelization in our approach. We have chosen Hwang and Saltz's method [7] for identifying induction pointers in a program, based on the calculation of def-use chains of statements that construct and traverse recursive data structures. As commented above, we have already added def-use chains generation support within our framework, so including this method comes as a straightforward addition.

## 5 Related Work

In the past few years pointer analysis has attracted a great deal of attention. A lot of studies have focused on stack-pointer analysis, like [8] and [9], while others, more related to our work, have focused on heap-pointer analysis, like [10] and

[11]. Both fields require different techniques of analysis. Unfortunately, heap-pointer techniques have failed to achieve aggressive optimizations. We think this is partly caused by techniques being isolated from other complementary pointer analysis techniques.

We are particularly fond of the work by Sagiv et al. [4], [12]. Their use of abstract interpretation/abstract semantics, along with materialization, have been adapted for the development of our framework. It is worth noting though, that their analysis is much more costly, meaning they can only analyze simple operations over singly-linked lists. Otherwise, analysis times and memory use become prohibitive. Also, their technique is only able to correctly analyze simple structures because they lack the support to handle structures like doubly-linked lists or heterogeneous trees. In our approach we have strived and achieved to obtain suitable graph abstractions for this kind of data structures. Besides, we think the analysis run at manageable times for such a complex technique. Finally, it should be noted that our technique is able to analyze structures based in pointer arrays, which is unheard of for a shape analysis technique, as far as we know.

We have been inspired for this work by existing research compiler frameworks: Polaris [13], which permitted the development of some noteworthy optimizations in array-based Fortran programs; ORC [14], which covers the whole compilation process and targets Itanium processors; SUIF [15], used by many researchers to implement their compiler techniques; or Soot [16], that features different modules for bytecode optimizations in Java programs. Polaris and SUIF ended their life cycle, ORC and Soot seem to concentrate on low level optimizations, and none of them focuses primarily on pointer analysis. Our plan is not to outdo these long established frameworks, but to swerve in a more specific direction where there is still plenty of room for optimizations related to parallelism and locality.

## 6 Conclusions and Future Work

As main contribution, we have introduced how a detailed shape analysis technique can be a valuable tool within a pointer analysis framework. Such a framework can combine different techniques towards better exploitation of parallelism. We have presented some early experiments that prove that shape analysis can be greatly improved when combined with information derived from other pointer analysis techniques, namely def-use chains.

We have also added support for the SSA form and def-use chains in Cetus. This support is useful in three ways: first, it helps to identify the statements that must be analyzed for correct shape analysis; second, it allows for automatic induction pointer recognition in the context of pointer-chasing loops, a key instrument for finding parallelism in recursive data structures; third, it allows for easy implementation of many pointer techniques that require SSA and/or def-use chains, enhancing the possibilities of the framework.

Only an early view of the pointer analysis framework has been presented. Still much work is needed to implement more pointer analysis techniques and make them work together towards finding unexploited parallelism in pointer-based programs. Also, we plan to conduct more experiments with benchmarks programs to fully test the capabilities of the techniques implemented.

## References

1. Johnson, T.A., Lee, S.I., Fei, L., Basumallik, A., Upadhyaya, G., Eigenmann, R., Midkiff, S.P.: Experiences in using Cetus for source-to-source transformations. In: The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04), West Lafayette, Indiana, USA (2004)
2. Tineo, A., Corbera, F., Navarro, A., Asenjo, R., Zapata, E.: Shape analysis for dynamic data structures based on coexistent links sets. In: 12th Workshop on Compilers for Parallel Computers, CPC 2006, A Coruña, Spain (2006)
3. Plevyak, J., Chien, A., Karamcheti, V.: Analysis of dynamic structures for efficient parallel execution. In: Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC'93). (1993)
4. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM Transactions on Programming Languages and Systems **20(1)** (1998) 1–50
5. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. In: ACM Transactions on Programming Languages and Systems (ACM'91). (1991) 13(4): 451–490
6. Tineo, A., Corbera, F., Navarro, A., Asenjo, R., Zapata, E.: A novel approach for detecting heap-based loop-carried dependences. In: The 2005 International Conference on Parallel Processing (ICPP'05), Oslo, Norway (2005)
7. Hwang, Y.S., Saltz, J.: Identifying DEF/USE information of statements that construct and traverse dynamic recursive data structures. In: Lecture Notes in Computer Science. Volume 1366 of Languages and Compilers for Parallel Computing (LCPC'97 Issue). (1998) 131–145
8. Hind, M., Pioli, A.: Which pointer analysis should I use? In: Int. Symp. on Software Testing and Analysis (ISSTA '00). (2000)
9. Wilson, R., Lam, M.: Efficient context-sensitive pointer analysis for C programs. In: ACM SIGPLAN'95 Conference on Programming Language Design and Implementation, La Jolla, CA (1995)
10. Ghiya, R., Hendren, L.J.: Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In: 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida (1996)
11. Chase, D., Wegman, M., Zadek, F.: Analysis of pointers and structures. In SIGPLAN Conference on Programming Languages Design and Implementation (1990) 296–310
12. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems (2002)

13. Blume, W., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, W., Rauchwerger, L., Tu, P., Weatherford, S.: Parallel programming with Polaris. IEEE Computer **29(12)** (1996) 78–82
14. Wu, C., Lian, R., Zhang, J., Ju, R., Chan, S., Liu, L., Feng, X., Zhang, Z.: An overview of the Open Research Compiler. In: The 17th International Workshop on Languages and Compilers for Parallel Computing (LCPC '04). (2005) 17–31
15. Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: An infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Notices **29(12)** (1994) 31–37
16. Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E., Co, P.: Soot - a Java optimization framework. In: Proceedings of CASCON 1999. (1999) 125–135