Discovering Optimistic Data-Structure Oriented Parallelism

Romain Cledat* *Intel Labs*
Santosh Pande *Georgia Institute of Technology*

Abstract

The parallelization of algorithms depends in large part on the understanding of data-access patterns. Regular data-structures, such as dense arrays, make pattern analysis easier as data-dependence graphs can be built to clearly identify computations that can happen in parallel. However, an important class of algorithms, such as machine learning and search, rely on "irregular" data-structures which make heavy use of pointers (trees, sparse graphs for example). While regular data-structures have well-defined properties with regards to their layout in memory, pointers obfuscate this.

In this paper, we argue that structure can also be found for irregular data-structures but at a different level of abstraction: the *symbolic* one. For example, each step of a breadth-first traversal of a binary tree on a node '`n`' will visit its '`left`' child pointer, followed by its '`right`' child pointer. While the relationships between the memory addresses of '`n`' and those pointed to by '`left`' and '`right`' may not exhibit a pattern, there is a definite relationship between the symbolic names '`n`', '`n::left`' and '`n::right`', where '`::`' denotes a parent-child relationship. These relationships can be used to compute data-dependence graphs just in time and therefore be able to help in determining whether two operations can run in parallel.

We present a trace-driven approach capable of identifying such relationships and motivate how the information could be used, for example, to improve optimistic parallel execution (such as STMs) as demonstrated by Cledat et al. in [3].

1 Introduction

Parallelism has moved from its traditional bastion of high-performance computing (HPC) to desktops, laptops and even mobile devices; the ARM Cortex-A9 [1] is a mobile multi-core chip for example. While researchers have had years to understand and optimize for the parallelism patterns of regular data structures such as dense arrays which are commonly used in HPC, emerging areas such as machine learning and search make use of more irregular data structures. An important characteristic of these algorithms is that the exact elements, and therefore memory locations, they access are heavily data-dependent and cannot be easily known until runtime. This cripples potential static analyses such as those used to efficiently parallelize dense matrix computations. However, these algorithms can still benefit from parallelization [9] and it is thus important to consider them.

*This work was done when the author was a PhD student at the Georgia Institute of Technology.

1.1 Key to parallelism: the computation's dataspace

A computation can be defined as being composed of **i)** an operation and **ii)** a *dataspace* which is the memory space the operation reads or writes. Understanding the dataspace of a computation is crucial to determining whether two computations A and B can execute in parallel because the *extent* of the dataspace, or union of all physical memory locations accessed by the computation, can lead to the determination of the amount of overlap between A and B: they can be parallelized if no overlap exists. Mendez-Lojo et al. understood the importance of this view in [8] where they stress the importance of a *data-centric* view of a computation. They contend that instead of thinking about dependencies between operations, one must take a view that encompasses the actions of the operations on the data.

In dense matrix operations, the extent of an operation's dataspace can frequently be computed statically as it is based on indices. A compiler can sometimes reason directly about the ranges of indices or a straightforward runtime function can be evaluated to determine the dataspace's extent.

Approaches for irregular algorithms The memory layout of irregular data-structures is, by definition, harder to characterize and this makes determining a computation's dataspace difficult. Current approaches punt on the problem in various ways. In [3], Cledat et al. present a framework to allow the programmer to directly specify properties that allow an approximate extent to be computed. In the Galois model [6, 8, 10], an expert programmer writes underlying data structures which a programmer can use in conjunction with smart design patterns to reduce the overhead of optimistic parallelism. The Galois model does not, however, a-priori estimate the extent of the dataspace although its 'one-shot' optimization is similar since it determines all the reads before doing all writes together.

Exploiting relationships The previous approaches discussed attempt to determine the computation's dataspace but rely on programmer knowledge (either predicates in [3] or algebraic properties in [8]. In this paper, we propose a *tracing approach* that seeks to determine statistical relationships between program variables. These relationships are then used to determine key program variables from which a computation's dataspace can be determined *before* the computation starts based on the runtime values of these variables.

Our approach is heuristical in nature and serves to guide the programmer in understanding the relationships between variables. One possible use of our approach, which we detail in this paper, is using the information generated to impact transaction scheduling in STM-like systems so as to minimize the probability of conflict. Correctness is not affected as the STM system will roll-back in the case of a conflict but performance can be increased as the likelihood of a conflict drops.

An important characteristic of our tracing approach is that we directly trace relationships between *program variables* as opposed to the hard-to-analyze physical memory addresses.

1.2 Contributions

This paper presents our symbolic[1] dataspace tracing framework and shows how the data gleaned can be analyzed and used to generate functions that can *predict* future memory accesses. This allows for the just-in-time computation of the extent of a computation's dataspace. We make the following specific contributions:

- We recognize the need for a more data-centric approach to concurrency focusing on extracting the relationships that exist in an operation's dataspace.
- We propose using the *symbolic* dataspace of an operation as opposed to its *address* dataspace to understand the relationships.
- We develop a memory tracer in the symbolic dataspace.
- We validate our framework on a graph coloring example showing how the analyzer is capable of determining the symbolic dataspace of an operation.

The remainder of this paper is organized as follows. Section 2 describes the differences between the address dataspace and the symbolic dataspace. Section 3 details our framework. Section 4 presents our experimental results, Section 5 reviews related work, in particular shape analysis,and we conclude with future work in Section 6.

2 Address dataspace versus symbolic dataspace

```
for(int i=0, e=end; i < e; ++i) {
  dataArray[i] = copyArray[i+10];
}
```

Figure 1: Example of a simple loop for a dense array where memory access patterns are fully predictable at the start of the loop

In dense array computations, data access patterns can frequently be understood statically at compile time by analyzing both the indices and the boundary conditions (loop bounds) used to access data. In the simplistic example loop shown in Figure 1, a compiler can predict the exact memory locations that will be accessed at the start of each loop iteration. These locations are solely determined by: **i)** the address of `dataArray`, **ii)** the address of `copyArray` and **iii)** the value of `i` which can all be determined at the start of each loop iteration. In this example, the memory locations will be known offsets off of the addresses of `dataArray` and `copyArray`. Therefore, the entire dataspace of the loop can be *predicted* with just the knowledge of `dataArray` and `copyArray` at the start of the loop. Determining if two such loops can execute in parallel in simply a matter of comparing these addresses to determine if there is any overlap.

In irregular programs, this type of analysis is rendered impossible by the heavy use of pointers which make static offset computations irrelevant. In essence, the physical layout of memory loses its importance for irregular programs.

2.1 Reachability and predictability

We seek to be able to predict an operation's dataspace before the operation actually executes. In Figure 1, prediction was made possible because all memory locations accessed were at known offsets from `dataArray` and `copyArray`. We will say that they were *reachable* from `dataArray` and `copyArray` where we define *reachable* as follows:

Definition An object B is said to be *reachable* from another object A if knowledge of the address of A gives knowledge of the address of B at runtime.

Reachability is key to predictability: if data elements read or written by an operation are all *reachable* from another common element R, then knowledge of R enables the determination of the operation's dataspace. For regular algorithms, objects are "reachable" from one another through the addition of a fixed offset and these relationships between objects are still visible in the address dataspace. Objects in irregular algorithms however are related to one another through pointers and this information is very difficult (if not impossible) to retrieve in the address dataspace. We seek to re-establish reachability arguments for irregular algorithms.

2.2 The symbolic dataspace

We define the symbolic dataspace of a program as follows:

Definition The symbolic dataspace of a program is a mapping between program variables and textual representations for these variables. In its simplest form, the program name of a variable can be used as its textual representation.

[1]Symbolic here refers to tracing program variable names as opposed to their address

In the symbolic dataspace, variables are thus associated with a programmer specified textual representation whereas in the address dataspace, they are associated with their address. Given that the textual representation is not arbitrary, it is possible to use it to encode relationships between variables in a way that includes relationships expressed through pointers.

We use the separator '`::`' to indicate a 'reachable' relationship. In other words, the textual representation '`n::left`' means that the memory location of '`n::left`' can be known from that of '`n`'. It does not matter if the relationship is through a fixed offset or a pointer as this is irrelevant from the point of view of reachability. The tracer we propose in this paper is capable of determining the symbolic names associated with every load and store in a program thereby making the 'reachable' relationships—which were lost at the memory level— visible. Note that the relationship between program variables and textual representations is not a one-to-one relationship due to aliasing (ie: the same variable may be accessed through various pointer indirection). The framework we develop can list all textual representations for a given program variable[2].

2.2.1 Motivating example

Consider the sample code in Figure 2. The presence of

```
struct Node {
 int data;
 Node *left, *right;
};
vector<Node*> nodes;
/* nodes is initialized to contain some
    Nodes */
for(int i=0, e=nodes.size(), i<e; ++i) {
 Node *n = nodes[i];
 if(n->left && n->right)
  n->data = n->left->data +
      n->right->data;
 else
  n->data = 0;
}
```

Figure 2: Example of a simple loop for a tree-like structure where memory access patterns are not obvious at the beginning of each iteration

the pointers '`left`' and '`right`' makes the addresses of '`n->left->data`' and '`n->right->data`' impossible to predict. In particular, there is no fixed offset between the addresses of '`n->data`' and '`n->left->data`'.

[2]The framework accounts for recursion and will properly list all representations that are non-recursive in nature.

In the symbolic dataspace, however, the access patterns are predictable and consistent: each iteration will access '`n::data`', '`n::left::data`' and '`n::right::data`'. The textual representation in the symbolic dataspace therefore allows for the same type of analysis as the ones possible in the address dataspace for regular computations.

The tracing approach we will detail in the following section coupled with analysis, will make explicit these relationships and therefore give the programmer information similar to that enjoyed for regular programs.

3 Symbolic dataspace analysis

Section 2 revealed the usefulness of describing access patterns in the symbolic dataspace. This section details a framework capable of tracing memory accesses in that dataspace. Our tracer is very similar to a traditional memory tracer which indicates the memory locations that are touched except that it outputs enough information for an analyzer to be able to interpret those memory locations in terms of symbolic names instead of memory addresses. For a given memory access, the analyzer will be able to determine the set of *names* that refer to that memory address. Note that a location may have multiple names due to pointer aliasing. Since the names implicitly express reachability, it will then be possible to estimate the 'root' variables that predict the dataspace (similar to `dataArray` and `copyArray` in Figure 1. Note also that our analyzer deals with cycles by "unrolling" the cycle as much as it can (ie: without forming a cycle). The analyzer will therefore always output a complete but bounded set of names for each memory location.

3.1 Components of the tracer

We define the following terms:

- **The focus area** is a section of code that the programmer wishes to analyze in terms of access patterns. Typically, a focus area will be a transaction (in STM language). The goal of the framework is to be able to determine the variables that can be used to predict the focus area's memory footprint at the start.
- **The location name** is a programmer-defined name associated with a physical memory location. Multiple names may be associated with the same location (due to aliasing) and the association may change over time.
- **The semantic memory map** is the mapping between a set of location names and physical memory addresses. In other words, it is a mapping between the semantic dataspace and the address dataspace. We will denote it SM_i where i is an instruction count as this mapping changes over time. Note that this mapping is only valid *inside* a single thread as instruction counts are not synchronized across threads. This is, however, sufficient for our analysis as we are interested in determin-

ing the dataspace of a computation before that computation starts based on the values and addresses of variables known at the start of the computation. We therefore only need to extract relationships between variables *within the same thread*. The ranges of addresses determined for each computation to launch in parallel can then be compared, at runtime, to determine if there is any possible overlap.

The framework is divided into three parts:

- **C++ template wrappers** are provided to the programmer to associate a *name* with program variables that he wants to track. These variables are typically the ones that are accessed in parallel regions of code and are therefore potentially shared. An automated way to associate names with variables could also be devised. The C++ API also provides mechanisms to identify the operations that are of interest to the programmer, typically the transactions themselves (in STM terminology).
- **A tracing compilation pass** is responsible for adding tracing annotations to the application. This is done in LLVM. When running, the application will dump a tracer file which is analyzed by the analyzer.
- **An analyzer** takes as input the information dumped during the application's execution and builds the mapping between physical memory and semantic names during the execution of the program. The current analyzer provides an interactive environment to give the programmer insight into the symbolic dataspace access patterns; it is currently not fast enough to be used JIT.

The role of the tracing pass is therefore to collect enough information so that the analyzer can correctly build and maintain $\mathrm{SM}_i \forall i$.

3.2 Operating principle

The tracer will collect information that affects SM as well as the addresses of loads and stores. During analysis, the addresses of the loads and stores can be mapped back to a set of names using SM. Note that the names associated with each memory location are *dynamically constructed*. For example, suppose a variable of type '`Node`' has a child pointer '`left`'; at compile time, the exact name of the node may not be known but at runtime, it can be determined (say to be '`n1`') and therefore the name of the memory location pointed to by '`left`' will be named '`n1::left`'.

The tracer will track:

- Malloc/New as they create a mapping between a physical location and a name.
- Free/Delete as they remove such a mapping.
- Pointer arithmetic as they modify a mapping by associating a name with a different physical location
- Other memory operations such as `memmove` or `memcpy` as they also alter the mapping.

Note that to minimize the tracing overhead, information is only collected on operations on the variables wrapped with the C++ API. In particular, only those loads and stores that may be relevant to the wrapped variables are traced.

```
  struct Node {
2   TRACKED(int data, ‘‘::data’’);
    TRACKED_PTR(Node *left, ‘‘::left’’);
    TRACKED_PTR(Node *right, ‘‘::right’’);
  };

7 TRACKED(Node n, ‘‘n’’);
  TRACKED(Node n2, ‘‘n2’’);
  n->left = &n2;
```

Figure 4: Sample code segment

Illustration Consider the code segment[3] in Figure 4 to illustrate how SM is constructed. SM at the end of the code segment is shown in Figure 3 The association of the name '`n`' with the memory region $[0\mathrm{x}10; 0\mathrm{x}24]$ is caused by Line 7 and the association of '`n2`' with $[0\mathrm{xA}0, 0\mathrm{xB}4]$ by Line 8 [4]. The names of the children fields of '`Node`' are given at Lines 2 through 4. Note that the use of the leading '`::`' indicates that the name of the parent should be dynamically determined at runtime. We see here two types of notions of reachability: '`data`' is reachable through an offset from the address of its parent and '`right`' is reachable through pointer indirection. The former also present in regular data-structures while the latter is specific to irregular ones. Line 9 creates the relationship between '`n::left`' and '`n2`'. The names associated with memory address $0\mathrm{xA}0$ are thus '`n2::data`' and '`n1::left::data`' which express that the address $0\mathrm{xA}0$ can be reached from '`n2`' using one "hop" and from '`n1`' using two "hops".

3.3 Using the analyzer

Our analyzer currently allows the programmer to interactively view SM_i and in particular to determine the names at the beginning of each focus area from which all other memory accesses are reachable. Using this information, the programmer can then devise functions that can be run at the start of the operation to determine the extent of the operation's dataspace. In particular, this would allow the automatic generation of the `isDisjoint` functions as defined by Cledat et al. in [3].

[3]This is a faithful representation of the way the C code would look in our framework.

[4]Memory addresses were chosen arbitrarily.

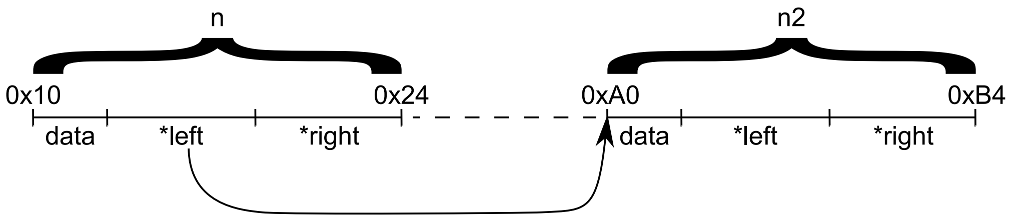

Figure 3: Memory map constructed by the analyzer at Line 9 of the code segment shown in Figure 4

4 Experimental validation

We validated our framework on a simple computation: a graph coloring example. Although this example is well known and no novel information will be gleaned from the analyzer, it allows us to validate our concepts and prove that the information collected is correct. Note that our framework is still not mature enough to be used in a JIT manner; the experimental validation presented here serves to illustrate our concepts.

4.1 Graph coloring

In graph coloring, each node of a graph must be "colored" in such a way that no two adjacent nodes are of the same color. This is commonly used for register allocation for example where the "colors" represent the various physical registers available and the nodes represent the virtual registers. An edge between two virtual registers means that they are live at the same time (and thus cannot be assigned to the same physical register). The workhorse function of the algorithm tries to assign a color to a node by looking at the colors of all adjacent nodes. This code is shown in Figure 5. Not shown in this code segment is the fact that each input node to the function is dynamically associated with the name '`input_node`'. Given the simplicity of the code, it is apparent that knowledge of '`input_node`' allows the full determination of the memory dataspace of the function[5]. Furthermore, there is only one write access which is to '`input_node::color`' For more complex codes however, this information may not be readily visible to the programmer.

4.2 Analyzer results

Running the tracer with a random graph of 10 nodes produces 5663 events (loads and stores as well as events related to building SM). Feeding this trace to the analyzer then opens an interactive environment which allows the programmer to set "weight" functions to rank location names based on their predictive power of future memory accesses. Several parameters can enter this formula but two in particular are of interest to us:

[5]'`graph`' is also used only to read in its size.

```
void processNode(Node* node, Graph*
    graph) {
  // The used vector contains which
      colors are used.
  vector<bool> used(graph->nodes->size(),
      false);
  for (unsigned int j = 0; j <
      node->adjacent_to->size(); ++j) {
    int color =
        *(node->adjacent_to[j]->color);
    if (color != -1) used[color] = true;
  }
  unsigned int smallest_color = 0;
  while (smallest_color <
      graph->nodes->size() &&
      used[smallest_color] == true)
    ++smallest_color;
  node->color = smallest_color;
}
```

Figure 5: Graph coloring code to process each graph node

- The type of memory access: whether it is a read or a write. This is important as concurrent reads are frequently permissible in parallel sections of code. In the formulas given later, r takes the value of 1 for read accesses and 0 otherwise. w similarly takes the value 1 for write accesses and 0 otherwise.
- The number of "hops" required to determine the address of the accessed location from the location name. This is important as the cost of determining the dataspace will be in part determined by how much pointer hopping is required. The variable d represents this number of "hops".

The interactive environment therefore allows the programmer to determine which location names allow for maximum prediction of the operation's dataspace at the start of the operation. This information can be used by the programmer to devise functions to execute before scheduling a `processNode` computation to determine the potential for overlap with other running `processNode` computations.

Determining write accesses Setting the formula to $\frac{w}{d*1000+1}$—which instructs the analyzer to only count write accesses and strongly favor location names that require few hops—returns that the only consistent write access across all ten iterations (one iteration per node) of the operation is '`input_node::color`'. For each individual iteration, the analyzer also properly identifies the specific node whose '`color`' field is accessed. With this information, the programmer could easily generate a function to run at the beginning of each iteration to determine the extent of the write dataspace; this function would simply compute the address of '`node->color`'. Note again that while this is trivial in this simple example, the tool would be just as useful for a complex program.

Determining read accesses Similarly, if the formula is set to $\frac{r}{d*1000+1}$ which only counts read accesses similarly favoring location names that require few hops, the analyzer reports that most of the read accesses are accessible from '`input_node::neighbors`' followed by '`graph::nodes`'. This corroborates what is expected in the sense that the function reads the color field of all of the neighbors of '`input_node`' as well as the size field of '`graph::nodes`'.

5 Related work

The work we propose is similar to other areas of research which we describe here.

5.1 Shape analysis

Shape analysis [4] aims to determine more accurate "points-to" information. In doing so, shape analysis also employs notions such as "reachability" [12] and "access paths" [7] which are very similar to the notions we present in this paper. In [5], Ghiya et al. even used shape analysis in a manner similar to ours to detect three common types of parallelism.

Our work, while similar to these works and others, differs in that our approach is based on a combination of static analysis and tracing as opposed to solely being based on static analysis. As a result, our approach is potentially less accurate (as it may not "see" certain accesses during its tracing) but also faster and potentially more precise as its analysis is only based on actual memory accesses. Our approach is thus not appropriate when a conservatively exact data footprint is required but is applicable for systems where a fast but potentially unsafe approximation of the footprint is allowed (such as STM-like systems).

We also believe that combining advanced static analysis techniques such as shape analysis and the symbolic tracing approach we describe in this work could lead to more precise estimations of data footprints.

5.2 Concolic execution

Concolic execution (which interleaves both concrete execution and symbolic execution) is a software verification technique that also considers program variables in a symbolic space. [13, 14] are examples of this where concrete inputs are computed so as to maximize code coverage.

5.3 Auto-parallelization methods

Much work has also gone into determining whether two sections of code are parallelizable [11, 2, 8, 10] among others, but our analysis is novel in its tracing approach and in mapping actual memory accesses with names which express relationships between variables. We are not aware of work that similarly tries to extract memory access patterns from names. We believe that our approach could enhance previous approaches.

6 Conclusion

We presented a tool aimed at memory tracing in the symbolic dataspace. We showed how this could be useful in determining a computation's dataspace for irregular algorithms. We believe that memory tracing in the symbolic dataspace, which makes relationships between variables explicit, could have more uses in parallel computing as well as in other areas of of computing. The patterns that are made visible in the symbolic dataspace could, for example, be used to classify algorithms and match them to well-known parallelism constructs. The current version of our tool is limited in the size of the program it can analyze due to the high memory consumption of the analyzer. We plan to couple the analyzer with a database backend to be able to quickly store and retrieve relevant patterns and relieve memory pressure. We then plan to release the tool to the community. Another possible extension of our work is to make it fast enough to be able to execute at runtime which would allow an optimistic just-in-time parallelization of code.

Acknowledgment

The authors would like to thank the anonymous reviewers for their comments which improved this paper. This research was done when Romain Cledat was a PhD student at the Georgia Institute of Technology and the authors would like to gratefully acknowledge the support of NSF grants CCF-1018544 and CCF-0916962 which supported this research.

References

[1] ARM. Cortex-a9 processor. `http://www.arm.com/products/processors/cortex-a/cortex-a9.php`.

[2] R. L. Bocchino, Jr., V. S. Adve, D. Dig, S. V. Adve, S. Heumann, R. Komuravelli, J. Overbey, P. Simmons, H. Sung, and M. Vakilian. A type and effect

system for deterministic parallel java. In *OOPSLA '09*, pages 97–116, New York, NY, USA, 2009. ACM.

[3] R. Cledat, K. Ravichandran, and S. Pande. Leveraging data-structure semantics for efficient algorithmic parallelism. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 28:1–28:10, New York, NY, USA, 2011. ACM.

[4] A. Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 230–241, New York, NY, USA, 1994. ACM.

[5] R. Ghiya, L. Hendren, and Y. Zhu. Detecting parallelism in c programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1:35–47, 1998.

[6] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew. Optimistic parallelism requires abstractions. In *PLDI '07*, pages 211–222, 2007.

[7] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[8] M. Mendez-Lojo, D. Nguyen, D. Prountzos, X. Sui, M. A. Hassan, M. Kulkarni, M. Burtscher, and K. Pingali. Structure-driven optimization for amorphous data-parallel programs. In *PPoPP '10*, New York, NY, USA, 2010. ACM.

[9] S. S. Mukherjee, S. D. Sharma, M. D. Hill, J. R. Larus, A. Rogers, and J. Saltz. Efficient support for irregular applications on distributed-memory machines. In *PPOPP '95*, pages 68–79, New York, NY, USA, 1995. ACM.

[10] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 12–25, New York, NY, USA, 2011. ACM.

[11] M. C. Rinard and M. S. Lam. The design, implementation, and evaluation of jade. *ACM Trans. Program. Lang. Syst.*, 20(3):483–545, 1998.

[12] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, pages 105–118, New York, NY, USA, 1999. ACM.

[13] K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 571–572, New York, NY, USA, 2007. ACM.

[14] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005. ACM.