Implications of a Data Structure Consistency Checking System

Viktor Kuncak, Patrick Lam, Karen Zee, and Martin Rinard

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street, Cambridge, MA 02139, USA
{vkuncak,plam,kkz,rinard}@csail.mit.edu

Abstract. We present a framework for verifying that programs correctly preserve important data structure consistency properties. Results from our implemented system indicate that our system can effectively enable the scalable verification of very precise data structure consistency properties within complete programs. Our system treats both *internal* properties, which deal with a single data structure implementation, and *external* properties, which deal with properties that involve multiple data structures. A key aspect of our system is that it enables multiple analysis and verification packages to productively interoperate to analyze a single program. In particular, it supports the targeted use of very precise, unscalable analyses in the context of a larger analysis and verification system. The integration of different analyses in our system is based on a common set-based specification language: precise analyses verify that data structures conform to set specifications, whereas scalable analyses verify relationships between data structures and preconditions of data structure operations.

There are several reasons why our system may be of interest in a broader program analysis and verification effort. First, it can ensure that the program satisfies important data structure consistency properties, which is an important goal in and of itself. Second, it can provide information that insulates other analysis and verification tools from having to deal directly with pointers and data structure implementations, thereby enabling these tools to focus on the key properties that they are designed to analyze. Finally, we expect other developers to be able to leverage its basic structuring concepts to enable the scalable verification of other program safety and correctness properties.

1 Introduction

This paper discusses a set of issues that arise in the verification of sophisticated program correctness and consistency properties. The backdrop for this discussion is our experience building the Hob program analysis and verification system, which verifies that programs correctly preserve detailed data structure consistency properties. There are several reasons that this experience is relevant to a larger program analysis and verification effort. Data structures usually play a central role in the program. Other kinds of program correctness properties often depend on the data structure consistency properties. Analyses that are designed to verify other program correctness properties must therefore incorporate (and in some cases interact with) the analyses that verify data structure consistency properties. Failure to either verify data structure consistency

© IFIP International Federation for Information Processing 2008

properties or to present these properties in a form that supports further analysis can therefore threaten the entire program verification effort.

Data structure consistency properties are also some of the most challenging program properties to analyze and verify. Data structure consistency properties often involve complex relationships between pointers, arrays, and unbounded numbers of data objects. There is no consensus on an abstraction or analysis that would be suitable for effectively reasoning about such properties. Indeed, recent years have seen a proliferation of abstractions and analyses, each with an ability to support the verification of a particular class of data structure consistency properties [15, 2, 8, 25, 27]. It currently seems unlikely that any single approach will prove to be successful for the full range of data structures that developers will legitimately desire to use. Any system that overcomes these substantial difficulties to successfully verify detailed data structure consistency properties in non-trivial programs is therefore likely to provide concepts and approaches that will be relevant to other analysis and verification efforts. We see several specific contributions that our concepts, system, and overall approach can make to a broad program analysis and verification effort.

Data Structure Consistency Properties. Data structure consistency properties are important in and of themselves. Our system shows how to automatically verify detailed data structure consistency properties in complete programs. In particular, it shows how multiple analysis and verification systems can cooperate to verify a diverse range of properties.

Foundational System. Pointers and the data structures that they implement are a key complication that any analysis or verification system must somehow deal with. In many cases pointers are tangential to the primary focus, but if the analysis or verification system does not treat them soundly, the system can deliver incorrect results. One contribution of our system is that it provides a layer that encapsulates the pointers behind data structure interfaces and provides a characterization of the properties that objects accessed via pointers or retrieved from data structures satisfy. Our system builds on this layer, as can other systems, to obtain the data structure and pointer information needed to provide correct results.

Transferable Concepts and Approaches. Our framework provides several concepts and approaches that developers ought to be able to leverage when they build their analysis and verification tools. Approaches that we think will be relevant in other areas include 1) our approach for applying very precise, unscalable analyses to targeted sections of the program as part of a broader scalable analysis and verification effort and 2) our technique for eliminating specification aggregation (Section 2.3), which occurs when procedure preconditions propagate up the procedure call hierarchy to complicate the specifications of high-level procedures.

Multiple Interoperating Analyses. One of the major themes of this paper is the need for multiple analysis and verification systems to interoperate to analyze the same program. Attempting to build a single general system that treats all analysis and verification problems in a uniform way is counterproductive—it forces every potential developer to understand the system and work within it if they are to contribute and makes it difficult to combine results from different, potentially independently developed, program analysis and verification systems.

2 The Hob System

The Hob system is based on several observations about data structures and how systems use them.

Encapsulated Complexity. Many data structures are designed to provide efficient implementations of relatively simple mathematical abstractions such as sets, relations, and functions. Appropriately encapsulating the data structure implementation behind an abstraction boundary (as in an abstract data type) can effectively encapsulate this implementation complexity. The complexity of the data structure (and therefore the complexity of reasoning about its consistency properties) is substantially larger inside the implementation than outside the implementation. In particular, it is usually possible to completely encapsulate any use of pointers within the data structure implementation. This encapsulation eliminates the need for analyses of data structure clients to reason about pointers—they can instead simply reason about the mathematical abstraction that the data structure implements.

Internal and External Consistency Constraints. Most programs contain two kinds of data structure consistency constraints. *Internal constraints* identify properties of a single encapsulated data structure. These constraints typically deal with elements of the low-level representation of the data structure such as relationships between pointers and array indices. *External constraints*, on the other hand, involve multiple data structures and typically deal with individual data structures at the level of the mathematical abstraction that the data structure implements. A typical external constraint might, for example, state that one data structure contains a subset of the objects in another data structure.

Client Dependence. Many data structure implementations will violate their internal consistency constraints if their clients use them incorrectly. For example, a linked list implementation may corrupt its internal representation if asked to insert an object into the list that is already present. Any practical data structure consistency analysis must therefore analyze both data structure implementations and clients.

Diversity. Known data structures have a diverse range of internal consistency properties. Moreover, new data structures may very well come with new and unanticipated kinds of properties.

The overall design and approach of the Hob system takes these observerations into account and differs substantially from previous data structure analysis systems.

2.1 Decoupled Approach with Multiple Cooperating Analyses

In our approach, each data structure is encapsulated in a module, which consists of three sections: an implementation section, a specification section, and an abstraction section (which provides definitions for abstract specification variables). The *implementation*

```
spec module DLLIter {
  format Node;
  specvar Content, Iter : Node set;
  invariant Iter in Content;

  proc isEmpty() returns e:bool
    ensures not e <=> (card(Content') >= 1);
  proc add(n : Node)
    requires card(n)=1 & not (n in Content)
    modifies Content
    ensures (Content' = Content + n);
  proc remove(n : Node)
    requires card(n)=1 & (n in Content)
    modifies Content, Iter
    ensures (Content' = Content - n) & (Iter' = Iter - n);

  proc initIter()
    requires card(Iter) = 0
    modifies Iter
    ensures (Iter' = Content);
  proc nextIter() returns n : Node
    requires card(Iter)>=1
    modifies Iter
    ensures card(n')=1 & (n' in Iter) & (Iter' = Iter - n');
  proc isLastIter() returns e:bool
    ensures not e <=> (card(Iter') >= 1);
  proc closeIter()
    modifies Iter
    ensures card(Iter') = 0;
}
```

Fig. 1. Specification Section of a Doubly Linked List with an Iterator

section of a Hob module is written in a standard imperative language. The *specification section* of a module is written in terms of standard mathematical abstractions such as sets of objects. Each exported procedure has a precondition and postcondition expressed as first-order logic formulas in the language of sets. To illustrate the benefits of set interfaces, Figure 1 presents the specification section of a module implementing a doubly-linked list with an iterator. Note how complex manipulations of a list data structure are replaced by a relationship between the values of sets before and after procedure execution. The *abstraction section* is written in whatever language is appropriate for the analysis that will analyze the implementation. This section indicates a representation invariant that holds whenever control is outside of the data structure implementation, and provides the values of abstract variables (sets) in terms of the concrete variables (the values of fields of a linked data structure or expressions involving global arrays).

While this design adopts several standard techniques (invariants, the use of preconditions and postconditions to support assume/guarantee reasoning), it deploys these techniques in the context of very strong modularity boundaries that fully decouple the analyses. In particular, it is possible to apply different analyses to verify different data structure implementations and clients. Moreover, the complexity of each individual data structure implementation is encapsulated behind the data structure's interface. Here is how this design has worked out in practice.

Multiple Targeted Analyses. We have developed a variety of analyses, with each specific analysis structured to verify a specific, fairly narrow class of data structures. The ability to target each analysis to a specific class of data structures has provided substantial benefits. Eliminating the burden of building a single general analysis has reduced the overall development overhead and enabled us to produce very narrow but very sophisticated analyses with relatively little engineering effort. It has also reduced the amount of broad expertise any one person needs to acquire to develop an analysis. Finally, it has enabled us to simply decline to implement problematic special cases. These properties have made it much easier for us to bring people together to work on the system since the barrier to entry (in terms of required program analysis and verification expertise) to development effort for any one analysis are so much smaller.

Interoperating Analyses. We have been able to productively apply multiple cooperating analyses to the same program. This property has been absolutely crucial to developing a reasonable system in a reasonable amount of time—it has given us effective abstraction barriers that have allowed us to decouple individual development tasks and farm these tasks out to different people. This development strategy has had two key benefits: first, it has allowed us to parallelize the work, and second, it has allowed us to bring the strengths of multiple people to bear on the project, with each person given a task best suited to his or her capabilities.

Relief from Onerous Scalability Requirements. Because the data structure interfaces are written in terms of high-level mathematical abstractions (rather than implementation-level concepts such as pointers), the data structure implementation complexity remains encapsulated inside the implementation and is not exposed to the client. Of course, a data structure's implementation must be analyzed using some analysis technique. Because implementations may be arbitrarily complicated, and because our system aims to verify sophisticated data structure consistency properties, it is difficult to imagine any suitable analysis which could scale to sizable programs. However, our design eliminates any need for any single data structure analysis to scale—an analysis needs only analyze the data structure implementation, leaving the analysis of the clients to simpler and more scalable analyses.

Consider the implications of this approach. Roughly speaking, much of the history of program analysis deals with managing the trade-off between scalability and precision. To a first approximation, it is relatively straightforward to build an analysis or verification strategy for almost any property of interest if scalability is not a concern. It has also proved to be possible to build analyses of almost arbitrary scalability [26, 24] as long as precision is not a concern. Building scalable, precise analyses has, however, eluded the field despite years of effort. Our approach averts this problem by 1) limiting the amount of code that any one internal data structure consistency analysis is responsible for processing to the data structure implementation code, and 2) enabling the use of less precise, more scalable analyses outside of the data structure implementations.

The result is that we have been able to effectively use analyses whose scalability limitations would be prohibitive in any other context. Specifically, we have used analyses with exponential and super-exponential complexity [10] and even made good use of interactive theorem proving [30].

2.2 Clean Analysis Problems

One of the key problems that program analysis and verification researchers have struggled with is what abstraction to use for programs with pointers [5, 20, 15]. Indeed, this question is still open today and is the subject of much ongoing research. Standard approaches have used either special-purpose logics [18] or implementation-oriented adhoc formalisms such as graphs [23]. The result is that the field has been effectively estranged from many years of research into more standard mathematical foundations, which have provided a significant body of potentially useful results in areas such as set theory and more standard logics.

Our elimination of pointers as a concept outside of data structure implementations has enabled us to use more standard mathematical abstractions (sets and relations) for the majority of the program. This has, in turn, allowed us to effectively draw on the large body of research on the properties of these standard mathematical abstractions.

2.3 Specification Aggregation

During our development of the system we encountered a problem that, as far as we can tell, will complicate all attempts to use assume/guarantee reasoning to achieve modular program verification. Assume/guarantee reasoning starts with procedure preconditions and postconditions. To verify a procedure call, it translates the precondition into the caller's context, verifies that the analyses or verification fact at the point before the procedure call implies the translated precondition, then translates the postcondition into the caller context to obtain the analysis or verification fact at the point after the procedure call. It can verify that the procedure correctly implements its precondition and postcondition independently. In this way, assume/guarantee reasoning enables modular program analysis and verification.

If we attempt to apply this reasoning approach, however, we soon run into *specification aggregation*. To verify the precondition of the invoked procedure at a procedure call site, we typically have to include some form of the precondition in the precondition of the calling procedure. The preconditions therefore aggregate as we move up the procedure call hierarchy. At the top of the hierarchy the procedure preconditions and postconditions can become unmanageably complex. Moreover, the need to aggregate preconditions and postconditions violates the modularity of the program, as the preconditions of leaf procedures inappropriately appear in the preconditions of transitive callers; in principle, these transitive callers should be unaware of the low-level implementation details of the procedures that they invoke.

Our solution is to use aspect-oriented concepts to pull invariants out into specifications which exist on-the-side; such invariants live in *scopes* [11]. A scope identifies an invariant and the part of the program that may update the invariant. Because these invariants do not appear in procedure preconditions or postconditions, they do not cause specification aggregation. The analysis or verification algorithm does, however, have access to the invariant and can use it to prove properties anywhere except in the region of the program that may update the involved state. Scopes differ from hierarchical structuring mechanisms in that they can contain arbitrarily overlapping modules and avoid the dominant program decomposition problem. The scope construct works well with data structure consistency properties, since they tend to be true throughout most of the program and updated only in relatively small portions. The end result is a substantial simplification of the specification of the program.

2.4 Experience

We have built a prototype system and used this system to verify a range of data structure consistency properties [10,30,11,12]. As expected, we have been able to use unscalable analyses to verify very detailed internal data structure consistency properties. Specific properties include the consistency of linked data structures such as linked lists (both singly and doubly linked lists), trees, and array-based data structures. Our system is the first to verify such properties in the context of complete programs.

Our system has also been able to use the results of the analysis outside the data structure implementation to verify that the program uses the data structure correctly. In particular, we have also been able to use multiple analyses on the same program, then combine the analysis results to verify higher-level consistency properties that involve multiple data structures. These properties include correlations between data structures, for example that two data structures contain disjoint sets of objects. These properties often capture application-level constraints; for instance, in our Minesweeper program [10], we verify that the set of revealed cells is disjoint from the set of hidden cells.

Our system, perhaps surprisingly, enables developers to verify program correctness properties that may not appear, at first, to be data structure consistency properties. Specifically, we have been able to express typestate properties of objects and verify that programs do not invoke operations on objects when they are in the wrong typestate.

We have verified programs that are roughly one to two thousand lines long and contain multiple data structures analyzed by different analyses. Moreover, these programs implement complete computations such as the popular Minesweeper game, Water (a scientific computation that simulates liquid water) [3], and a web server. Our ability to demonstrate that our system is capable of verifying larger programs is limited largely by our ability to develop or port these programs.

3 Comparison to Some Related Approaches

Frameworks for formal software development use the idea of data refinement [7, Chapter 8] but achieve levels of automation similar to the use of our system with an interactive theorem prover alone [30]. The use of the full strength of our system provides a greater degree of automation compared to approaches based purely on verification condition generation and interactive theorem proving, thanks to the use of decision procedures and techniques for loop invariant inference. Like [7], our system acknowledges the importance of both aspects of the verification: the verification of data structure implementations and the verification of data structure clients. In contrast, most existing static analysis approaches verify only the clients of interfaces, typically expressed as finite state machines [1, 6], [22, Chapter 6]. The interfaces in Hob are more expressive than finite state machines, because they can express finite-state properties of an unbounded number of objects, and because they can express cardinality constraints on the number of objects that satisfy a given property. Researchers have also explored the verification of the usage of interfaces that are based on first-order logic [19]. Implementations of abstract data types have also been verified using TVLA [14]. Integration of these two sides—implementatations and interfaces—of verification in TVLA using assume/guarantee reasoning is the subject of ongoing research [29, 28]. Our approach in Hob was to single out the simple, yet powerful abstraction of global sets and explore the range of properties that such interfaces support [12]. Hob and the Spec# verifier [2] address different points in the design space. Whereas Hob adopts a simple model of encapsulation using modules and introduces new constructs for exploring novel overlapping inter-module grouping mechanisms such as scopes, Spec# uses instantiatable classes as the main unit of encapsulation and remains close to its starting point, the programming language C#. Regarding the level of automation, Hob appears to provide more automated handling of reachability properties in tree-like data structures, whereas Spec# has more support for arithmetic; these differences are partly a consequence of design decisions and partly a conequence of the decision procedures employed in these two systems. Finally, there is currently little emphasis on abstract specification variables in Spec#, whereas Hob uses them as the starting point for scalable analysis of the largest parts of the program.

4 Implications for Other Efforts

We see our system as relevant to other analysis and verification efforts in two ways. First, our treatment of pointers and data structures can serve as a foundation for other analysis or verification efforts that must deal somehow with programs that contain pointers and data structures. We envision analyses whose primary focus is not to verify detailed properties involving data structures or pointers, but that rely on the truth of some incidental data structure properties for the analysis to succeed. We envision our analysis providing these other analyses with a relatively abstract, tractable, and verified view of the data structures and pointers. Ideally, our system would give the developers of the new analysis or verification system the information they need quickly and easily, enabling them to productively focus their efforts on the problem of interest.

Second, we believe that the developers of other analyses may be able to use several of the concepts from our system to build analysis frameworks for their analysis problems. By building on these concepts, these analysis frameworks would be able to support the targeted application of multiple very precise, interoperating, unscalable analyses in a scalable way to a single program. We view our ideas as likely to be particularly useful when there is some relatively small part of the program that manipulates, in a fairly complex way, a clearly delineable part of the state (either of the program itself or of some system that it interacts with). Outside this small part of the program the state may be of interest but there is nothing complex going on. While data structures provide a canonical example of such a situation, we believe that this basic pattern is pervasive in modern software.

5 Future Work

We have implemented a prototype system Hob [13] for verifying data structure consistency and successfully applied it to a range of programs. Several further problems are worth exploring as we move forward; many of these problems are not specific to the domain of data structure consistency properties.

Specialized analyses and libraries of verified data structures. Among the strengths of our approach is the ability to verify a wide range of properties for a variety of data structures. This strength comes from the availability of specialized analyses for common data structures. Researchers have successfully verified many properties of tree-like data structures; on the other hand, there are fewer extant results on data structures that use arrays and non-tree-like data structures. Many ubiquitous data structures still lack verified implementations; we envision verifying them using techniques with varying levels of automation and building a library of verified data structures. We expect that, as such libraries grow, there will be many common reasoning patterns that will allow the results of verification to be extrapolated into fully automated analyses. Our approach supports such incremental development because it supports both interactive theorem proving and analyses with an increasing degree of automation.

Relevant tractable fragments of general-purpose logics. By using logical formulas to communicate analysis results, our system makes it convenient to build analyses that themselves use logic to encode dataflow information inside the implementations of modules. Such analyses are often precise and predictable because it is possible to describe the class of properties to which they apply. It is therefore useful to explore new classes of computationally tractable fragments of logics and constraints that can be used as a basis for analyses. We suggest defining these logics as fragments of general logics such as typed set theory, which have proven successful in formalizing a wide range of properties. The study of logical fragments allows us to deploy specialized algorithms while retaining simple semantics and the ability to communicate between different analyses. Our experience suggests that, although traditional classifications based on simple syntactic criteria are still useful [4, 9], data structure consistency constraints are likely to yield new kinds of classifications and new ways of defining subclasses of logics [17].

Experience from larger applications. Experience from using our techniques in the context of larger applications would further contribute to understanding the data structure consistency problem. We expect that the problem of internal data structure consistency is essentially the same in both large and small applications, with larger applications having greater diversity and wider data structure interfaces (to support many usage scenarios). We also believe that we have identified some of the high-level data structure consistency properties (such as disjointness and inclusion) that are likely to be generally useful. It remains to investigate classes of more complex high-level properties. It is possible that most of these properties will be domain-specific, with different kinds of useful and tractable constraints applicable to different domains.

Supporting common language features. To obtain experience with larger applications, it is important to support the features of commonly used programming languages. The evolution of languages has simultaneously contributed 1) features that simplify program semantics (such as memory safety and the ability to encode simple invariants using types) and 2) features that complicate reasoning (such as higher-order functions, continuations, dynamic dispatch, exceptions, reflection, and concurrency). An attempt to handle the worst-case scenario arising from the use of these features is not likely to be fruitful; it is instead important to consider the patterns in which these features are used and adapt the analyses to work reasonably well in these cases. In addition to making the automated analysis of these features practical, the study of these patterns is likely to yield important results in programming methodology and programming language design.

Correctness of analysis results. One of the major themes of this paper is the need for multiple analyses to interoperate on the same program. Ideally, implementors will have maximum flexibility in the implementation of these analyses, enabling the full range of implementors to bring their skills effectively to bear and make a contribution. In particular, we envision developers with varying areas of expertise, levels of competence, and programming styles and inclinations. Any time one combines the work of multiple people, questions of competence and trust arise. An error in one analysis or verification can call the entire result into question. We therefore believe that it is important to build a system that can verify the results of the various analyses and verifications. Such a system would accept and verify proofs of correctness of the results. We envision a system similar to Credible Compilation [16, 21] in which each analysis or verification system would generate, for each part of the program it processed, a proof that the specific result it generated on that analysis or verification is correct.

6 Conclusion

We are becoming ever closer to having the basic requirements in place for a successful and ambitious program analysis and verification project—a recognized and growing acknowledgement of the need for more reliable software, the raw computing power necessary to support the required reasoning, and a community of program analysis and verification researchers that, given an appropriate time and space budget, is able to deliver algorithms that extract or check virtually any well-defined property of interest.

Important remaining barriers include techniques that deal effectively with pointers and data structures and, especially, ways to bring multiple analyses together to interoperate during the analysis of a single program. It is especially important to support the targeted application of unscalable approaches in the context of a larger scalable analysis effort—these unscalable analysis and verification algorithms are the only way to verify the precise, detailed properties to which any successful analysis and verification effort must aspire.

We have addressed all of these issues in the context of the Hob system for verifying data structure consistency. This system provides an effective analysis interface for providing other analyses with pointer and data structure information. It has also employed a range of techniques that have enabled the successful coordinated application of a range of unscalable analyses to complete programs. These techniques, and especially the concepts behind them, should generalize to enable the construction of other systems for scalably verifying very precise program safety and correctness properties.

References

1. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic predicate abstraction of C programs. In: Proc. ACM PLDI (2001)
2. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
3. Blume, W., Eigenmann, R.: Performance analysis of parallelizing compilers on the Perfect Benchmarks programs. IEEE Transactions on Parallel and Distributed Systems 3(6), 643–656 (1992)
4. Börger, E., Grädel, E., Gurevich, Y.: The Classical Decision Problem. Springer, Heidelberg (1997)
5. Chase, D.R., Wegman, M., Zadeck, F.K.: Analysis of pointers and structures. In: Proc. ACM PLDI (1990)
6. Das, M., Lerner, S., Seigle, M.: ESP: Path-sensitive program verification in polynomial time. In: Proc. ACM PLDI (2002)
7. Jones, C.B.: Systematic Software Development using VDM. Prentice Hall International, UK (1986)
8. Kuncak, V., Lam, P., Rinard, M.: Role analysis. In: Annual ACM Symp. on Principles of Programming Languages (POPL) (2002)
9. Kuncak, V., Rinard, M.: Decision procedures for set-valued fields. In: 1st International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL 2005) (2005)
10. Lam, P., Kuncak, V., Rinard, M.: On our experience with modular pluggable analyses. Technical Report 965, MIT CSAIL (September, 2004)
11. Lam, P., Kuncak, V., Rinard, M.: Cross-cutting techniques in program specification and analysis. In: 4th International Conference on Aspect-Oriented Software Development (AOSD 2005) (2005)
12. Lam, P., Kuncak, V., Rinard, M.: Generalized Typestate Checking for Data Structure Consistency. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 430–447. Springer, Heidelberg (2005)
13. Lam, P., Kuncak, V., Rinard, M.: Hob: A tool for verifying data structure consistency. In: 14th International Conference on Compiler Construction (tool demo) (April, 2005)
14. Lev-Ami, T., Reps, T., Sagiv, M., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: International Symposium on Software Testing and Analysis (2000)
15. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Proc. 7th International Static Analysis Symposium (2000)
16. Marinov, D.: Credible compilation. Master's thesis, Massachusetts Institute of Technology (2000)
17. Marnette, B., Kuncak, V., Rinard, M.: On algorithms and complexity for sets with cardinality constraints. Technical report, MIT CSAIL (August, 2005)
18. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, Springer, Heidelberg (2001)
19. Ramalingam, G., Warshavsky, A., Field, J., Goyal, D., Sagiv, M.: Deriving specialized program analyses for certifying component-client conformance. In: PLDI (2002)
20. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th LICS, pp. 55–74 (2002)
21. Rinard, M., Marinov, D.: Credible compilation with pointers. In: Proceedings of the Workshop on Run-Time Result Verification (1999)

22. Rinetzky, N.: Interprocedural shape analysis. Master's thesis, Technion - Israel Institute of Technology (2000)
23. Sagiv, M., Reps, T., Wilhelm, R.: Solving shape-analysis problems in languages with destructive updating. ACM TOPLAS 20(1), 1–50 (1998)
24. Steensgaard, B.: Points-to analysis in almost linear time. In: Proc. 23rd ACM POPL, Petersburg Beach, FL (January, 1996)
25. Sălcianu, A.D., Rinard, M.: Purity and side-effect analysis for java programs. In: Proc. 6th International Conference on Verification, Model Checking and Abstract Interpretation (to appear, January 2005)
26. Xie, Y., Aiken, A.: Scalable error detection using boolean satisfiability. In: POPL 2005 (2005)
27. Yang, J., Twohey, P., Engler, D., Musuvathi, M.: Using model checking to find serious file system errors. In: OSDI 2004 (2004)
28. Yorsh, G., Reps, T., Sagiv, M.: Symbolically computing most-precise abstract operations for shape analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)
29. Yorsh, G., Skidanov, A., Reps, T., Sagiv, M.: Automatic assume/guarantee reasoning for heap-manupilating programs. In: 1st AIOOL Workshop (2005)
30. Zee, K., Lam, P., Kuncak, V., Rinard, M.: Combining theorem proving with static analysis for data structure consistency. In: International Workshop on Software Verification and Validation (SVV 2004), Seattle (November, 2004)

A Discussion on Patrick Lam's Presentation

Willem-Paul de Roever

I try to understand what you are doing, and you use a sequential setting, according to me.

Patrick Lam

Yes, that is right.

Willem-Paul de Roever

But then, can you explain, what the heck the assume-guarantee paradigm has to do with it, if you do everything sequentially?

Patrick Lam

What we mean by assume-guarantee is that we assume that the precondition holds, and then we show the postcondition. We are not talking about assume-guarantee as you usually talk about it in concurrent programs.

Willem-Paul de Roever

So, you have an individual interpretation. Thank you.

Richard Bornat

A comment rather than a question. You had a sentence that was approximately: Sets are intuitive for programmers. I say this with some trepidation in front of you [addressing Jean-Raymond Abrial] now: But there are only two people I know who are programmers and find sets intuitive. The other one is Bernard Sufrin, who developed Z with you [Abrial]. For the rest of us, sets are seductive, and we make the kind of nave errors that have caused so many problems historically in mathematics, that is, we mis-specify our programs, because we think sets are one thing, and they are in fact another. So, I plead with you: Please replace the word ”intuitive” with ”seductive”, and consider the consequences. Thank you.

Patrick Lam: OK.