Gregory Kulczycki et al.: Proving Correctness for Pointer Programs in a Verifying Compiler, Proceedings of The Sixth NASA Langley Formal Methods Workshop, p.59–61

Proving Correctness for Pointer Programs in a Verifying Compiler

Gregory Kulczycki, Amrinder Singh

Virginia Tech, Department of Computer Science, Falls Church, VA 22043, USA

{gregwk, amrinder}@vt.edu

Extended Abstract

This research describes a component-based approach to proving the correctness of programs involving pointer behavior. The approach supports modular reasoning and is designed to be used within the larger context of a verifying compiler. The approach consists of two parts. When a system component requires the direct manipulation of pointer operations in its implementation, we implement it using a built-in component specifically designed to capture the functional and performance behavior of pointers. When a system component requires pointer behavior via a linked data structure, we ensure that the complexities of the pointer operations are encapsulated within the data structure and are hidden to the client component. In this way, programs that rely on pointers can be verified modularly, without requiring special rules for pointers.

The ultimate objective of a verifying compiler is to prove-with as little human intervention as possible-that proposed program code is correct with respect to a full behavioral specification. Full verification for software is especially important for an agency like NASA that is routinely involved in the development of mission critical systems.

There are at least two fundamental problems in developing a verifying compiler that is scalable. One of these is modularity. There is near universal agreement that to be scalable, the verification system must be modular. In other words, it must be possible for the verifying compiler to take just the specifications of components used by a piece of code and to establish that the proposed implementation is correct with respect to its specification [1].

The other problem concerns the complexity of the assertions that are involved in the verification process. A variety of techniques have been explored in the literature to mitigate this. Most recent work involves "lightweight" formal methods, meaning a focus on specification-independent or easy-to-state ad hoc properties of an implementation. Such a system may be used, for example, to establish the absence of null dereferences [2] or to demonstrate the absence of cycles in a pointer-based data structure [3]. Lightweight methods offer two benefits: they relieve programmers from the need to write full behavioral specifications and internal assertions such as loop invariants, and they show that progress toward the goal of full verification can be incremental.

While the lightweight approach is necessary and useful in the immediate term, verifiers ultimately need to get beyond the point of showing merely that blatant error conditions do not exist, and to establish that programs actually achieve a full specification of desired behavior. A verifying compiler eventually must be able to deal with nontrivial assertions such as those needed to prove correctness of implementations that use pointers and references. Researchers who focus their attention on pointer verification problems tend to fall into two categories. Those who extend prior research in lightweight methods, and those who focusing on more general verification. Our work

falls into the second category.

From experience, we know that full verification is especially difficult for programs that involve pointers or references and linked data structures [4]. In some situations, pointers are unavoidable; in others, verification can be simplified by using suitable abstractions. Regardless, a verifying compiler should be able to handle both cases, preferably using the same set of rules. To enable this, we use a two-part approach to full verification of typical pointer programs.

The first part is used when pointer behavior is necessary to satisfy performance requirements. It replaces language-supplied pointers or references with a software component-which we typically refer to as the pointer component-whose specification abstracts pointers and pointer-manipulation operations [5]. The specification does not require any special techniques or constructs. The pointer component has a specification that allows programmers and, importantly, the verification system to view it as any other component. However, to achieve the same performance benefits of pointers, the compiler cannot implement it the same way it implements other components. For example, the call Relocate(p, q) assigns p to q's location, effectively resulting in p and q being aliases. Although the programmer can reason about the statement as a method call, the compiler will implement it by copying a single reference. By using the pointer component, we can ensure that the complications associated with indirection arise only from the specification of the pointer component itself. They occur only where the required behavior of the program to be verified relies on the need for indirection, rather than permeating all proofs of correctness.

We have used the pointer component in the verification of correctness for non-trivial algorithms such as the Schorr-Waite graph-marking algorithm [6]. In the case of the Schorr-Waite algorithm, we wrote the specification for it, implemented it with the pointer component, and showed that the implementation was correct with respect to the specification. This process led us to propose additional operations for the pointer component that simplified certain verification tasks. In particular, we can define and specify pointer component operations that do not create memory leaks. Using these operations in place of traditional ones in the implementation can ease the burden on the verification system.

The second part of our approach uses abstract specifications to encapsulate data structures, such as lists and trees, whose implementations typically require pointer behavior. This limits the pointer-associated verification complexity to the verification of the data structures themselves, and simplifies the verification of components that use these data structures. Since the data structures used in the second part of the approach often require pointer behavior, the pointer component described in the first part of our approach can be used to implement them. Regardless of whether we are verifying components implemented with the pointer component or component implemented with linked data structures, their full verification can be handled similarly. There is neither a need to focus on selected pointer properties, such as the absence of null references or cycles, nor a need for special rules to handle pointers.

Primarily, the language we have used in this research is Resolve, which is a combined programming and specification language designed to support modular verification [7]. It also supports full alias-avoidance at the component level. Efforts are being made to incorporate the programming aspects of our approach to other languages. For example, the Tako language is an object-oriented language with Java-like syntax that employs a pointer component and avoids many common sources of aliasing [8].

1. References

[1] Sitaraman, M., Atkinson, S., Kulczycki, G., Weide, B.W., Long, T.J., Bucci, P., Heym, W., Pike, S., Hollingsworth, J.E.: Reasoning About Software-Component Behavior. Procs. Sixth Int. Conf. on Software Reuse. Springer-Verlag (2000) 266-283

[2] Leino, K.R.M., Nelson, G., Saxe, J.B.: ESC/Java User's Manual. Compaq Systems Research Center (2000)

[3] Hackett, B., Rugina, R.: Region-Based Shape Analysis with Tracked Locations. POPL '05 (2005)

[4] Kulczycki, G., Sitaraman, M., Keown, H., Weide, B.: Abstracting Pointers for a Verifying Compiler. 31st Annual Software Engineering Workshop, Baltimore, MD (2007)

[5] Kulczycki, G.W., Sitaraman, M., Ogden, W.F., Hollingsworth, J.E.: Capturing the Behavior of Linked Data Structures. Proceedings RESOLVE 2002 Workshop (2002)

[6] Singh, A.: A component-based approach to proving the correctness of the Schorr-Waite algorithm. Computer Science, Vol. Master of Science. Virginia Tech, Falls Church (2007)

[7] Sitaraman, M., Weide, B.W.: Component-Based Software Using RESOLVE. ACM Software Engineering Notes 19 (1994) 21-67

[8] Kulczycki, G., Vasudeo, J.: Simplifying Reasoning about Objects with Tako. Proceedings of the Specification and Verification of Component-Based Systems Workshop, Portland, OR (2006)